INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67 D–80538 München

Ludwig
Maximilians–
Universität
München
LMU

A Visual Rendering of a Semantic Wiki Query Language

Andreas Hartl

Diplomarbeit

Beginn der Arbeit:	30.04.2009
Abgabe der Arbeit:	29.10.2009
Betreuer:	Prof. Dr. François Bry,
	Klara Weiand

Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 26.10.2009 Andreas Hartl

Abstract

KiWi is a semantic Wiki, aiming to combine the wiki philosophy of collaborative content creation and the methods of the Semantic Web to enable effective knowledge management.

Its intended use is in enterprise knowledge management and other social software settings, and thus encompasses a broad spectrum of possible users. To accommodate users with varying technological backgrounds and varying intended uses for the software, KiWi has its own keyword-based query language, KWQL, which allows for both very complex querying and construction based on the internal KiWi model, and simple keyword queries.

This thesis introduces visKWQL and the KWQL Query Builder, a visual rendering of and editor for the textual KWQL language, that allows the creation of queries by constructing them from visual building blocks, each corresponding to an operator or keyword of the KWQL language.

The Query Builder supports the user in the creation of queries by preventing operations that would result in syntactically incorrect queries, visually displays problems in the query, guides the user with hints and warnings through the query creation process and displays descriptions of the visible building blocks.

The Query Builder automatically translates the visual query into its textual form. Additionally, since it also includes a parser for KWQL, it supports round-tripping, allowing the user to edit the textual query without using the visual Query Builder, and see the changes reflected in the visual representation, or to further textually edit a query generated using the visual editor.

This thesis first gives an overview of different visual languages and methodologies and a short introduction to KiWi and the KiWi Query Language. Afterwards, visKWQL and the KWQL Query Builder are introduced, first giving an overview of the language and editor features, followed by a user's manual and a look at the implementation.

Zusammenfassung

KiWi ist ein semantisches Wiki, mit dem Ziel, die Wiki-philosophie der gemeinsamen Inhaltserstellung mit den Methoden des semantischen Webs zu kombinieren, um ein effektives Wissensmanagement zu ermöglichen.

Seine beabsichtigten Einsatzgebiete liegen im Enterprise Wissensmanagement und anderen sozialen Software Umgebungen, und umfassen damit ein breites Spektrum von potenziellen Nutzern.

Um Nutzern mit verschiedenem technologischen Hintergrund und verschiedenen Anforderungen an die Software gerecht zu werden, besitzt KiWi seine eigene Schlüsselwortbasierte Anfragesprache, KWQL, die sowohl sehr komplexe Anfragen und Konstruktionen auf Basis des internen KiWi Modells erlaubt, als auch einfache Schlüsselwort Suche.

Diese Diplomarbeit stellt visKWQL und den KWQL Query Builder vor, eine visuelle Darstellung der textuellen KWQL Sprache, zusammen mit einem Editor, der die einfache Erstellung von Anfragen durch Konstruktion aus einzelnen Bauteilen erlaubt, von denen jedes einem Operator oder Schlüsselwort von KWQL entspricht.

Der Query Builder unterstützt den Nutzer bei der Anfrageerstellung indem er Operationen verhindert, die zu syntaktisch inkorrekten Anfragen führen würden, indem er Probleme in einer Anfrage visuell darstellt, den Nutzer mit Tips und Warnungen durch den Prozess der Anfrageerstellung führt, und Erklärungen der einzelnen Bausteine anzeigt.

Der Query Builder übersetzt automatisch die visuelle Anfrage in die textuelle Version. Darüberhinaus, dank einem Parser für KWQL, unterstützt er auch “Round-Tripping”, was dem Nutzer erlaubt, die textuelle Anfrage ohne den Query Builder zu verändern, und zu sehen, wie sich diese Änderungen auf die visuelle Darstellung der Anfrage auswirken, oder den textuellen Output des Query Builders weiter zu bearbeiten.

Diese Diplomarbeit beginnt mit einem Überblick über diverse visuelle Sprachen und Methoden, und einer kurzen Einführung in KiWi und die KiWi Anfragesprache. Anschliessend werden visKWQL und der KWQL Query Builder vorgestellt, beginnend mit einem Überblick über die Sprache und die Fähigkeiten des Editors, gefolgt von einer Benutzeranleitung und einem Blick auf die Implementation.

Acknowledgments

My thanks go to my thesis supervisors *Prof. Dr. François Bry* and *Klara Weiand*, who greatly supported me in the writing of this thesis and the software, and contributed with many helpful comments and ideas.

Contents

1 Introduction **3**
1.1 Thesis Structure . . . 6

2 Overview of Visual Languages and Methodologies **7**
2.1 Reasons for Visual Language Research . . . 8
2.2 Visual Modeling Languages . . . 11
2.2.1 Structograms . . . 12
2.2.2 ER Diagrams . . . 14
2.2.3 UML . . . 17
2.3 Visual Programming Languages . . . 21
2.3.1 SIVIL . . . 23
2.3.2 VXT . . . 24
2.3.3 Yahoo! Pipes . . . 28
2.3.4 JBoss jBPM . . . 30
2.3.5 ComiKit . . . 32
2.3.6 3D-PP . . . 34
2.4 Visual Query Languages . . . 36
2.4.1 Form-based Visual Query Languages . . . 40
2.4.1.1 QBE . . . 40
2.4.1.2 EquiX . . . 42
2.4.1.3 BBQ . . . 43
2.4.1.4 Xing . . . 45
2.4.2 Diagram-based Visual Query Languages . . . 47
2.4.2.1 G . . . 47
2.4.2.2 GraphLog . . . 48
2.4.2.3 Good . . . 49
2.4.2.4 XML-GL . . . 52
2.4.2.5 visXcerpt . . . 55
2.4.2.6 NITELIGHT . . . 57
2.4.2.7 RDF-GL . . . 59
2.4.3 Hybrid Systems . . . 60
2.4.3.1 PESTO . . . 60
2.4.3.2 SKI . . . 61
2.4.3.3 IVQS . . . 63

3 A Visual Query Language for KiWi **65**
3.1 KiWi . . . 65
3.1.1 The KiWi Vision . . . 65
3.1.2 The KiWi System . . . 67
3.1.3 KiWi Usage Concepts . . . 68
3.1.4 Querying KiWi . . . 70
3.1.4.1 SPARQL . . . 70
3.2 KWQL . . . 72
3.3 visKWQL and the KiWi Query Builder . . . 75
3.3.1 Motivation and Design Goals . . . 75
3.3.2 Language and Editor Features . . . 78
3.3.2.1 Visual Formalism . . . 79
3.3.2.2 Interaction Strategy . . . 81
3.3.2.3 Roundtripping . . . 83
3.3.2.4 User Guidance . . . 83
3.3.2.5 Error Prevention . . . 85
3.3.2.6 Error Reporting and Correction . . . 88
3.3.2.7 Other Features . . . 90
3.3.3 Query Builder Usage . . . 92
3.3.3.1 The KQB User Interface . . . 92
3.3.3.2 (vis)KWQL elements . . . 94
3.3.4 Query Construction - User's Manual . . . 98
3.3.4.1 Creating Queries with Drag & Drop . . . 98
3.3.4.2 Advanced Examples . . . 102
3.4 Implementation . . . 104
3.4.1 Java Script . . . 105
3.4.2 Roundtripping . . . 105
3.4.3 Customizability . . . 107

4 Conclusion **109**

Bibliography **112**

Chapter 1

Introduction

In its early years, the content on the internet consisted mainly of static web sites, of personal or corporate homepages.

In recent years, with technological advancements that made it possible to create more interactive web sites, came a philosophy shift towards sharing, collaboration and user-centered design, the "Web 2.0". It gave rise to the now ubiquitous blogs, wikis, social networks and web applications.

Today, research for the next stage of the internet goes towards the Semantic Web.

While the Web 2.0 was designed around the human user, the aim of the Semantic Web is to make the web understandable to computers, by enriching the human-readable web content with computer-readable data, enabling automatic analysis, reasoning and knowledge discovery, and improved searching.

While the first stage of the web was built upon the Hyper Text Markup Language (HTML), and Web 2.0 upon the semi-structured language XML (Extensible Markup Language), the Semantic Web is characterized by its use of new and more complex information description methods and languages like RDF (Resource Description Framework) and OWL (Web Ontology Language).

KiWi [1] is an application, currently under development, for the Semantic Web.

Its aim is to combine knowledge management, the wiki philosophy of collaborative content creation and information sharing, and the methods of the Semantic Web, to create a framework for social websites for knowledge management in corporate and community environments.

Data in KiWi, be it the photos on a photo-sharing community website, or algorithm descriptions on an industrial software developer's customized KiWi system, have an underlying conceptual model consisting of wiki pages, textual fragments, links and annotations.

To search for data in KiWi, the KiWi Query Language KWQL [2] has been designed.

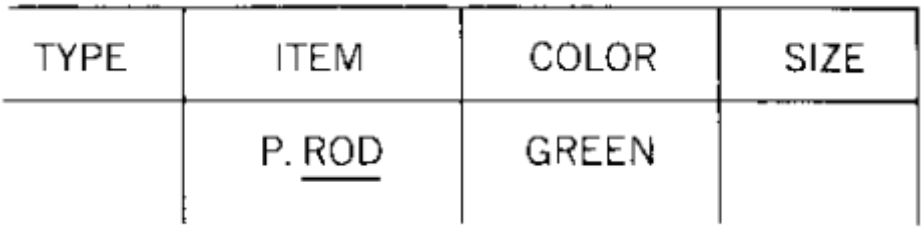
KWQL is a keyword-based query language, which combines the expressivity of traditional database query languages with the user-friendliness of search languages as they are used for internet search engines or to search traditional wikis. Thus, a KWQL query can be very complex and directly query data or metadata of KiWi's conceptual data model, but it can also consist of only a single keyword.

Due to its customizability, KiWi can be used in many different environments and for different purposes, which gives it a wide range of potential users with varying technological

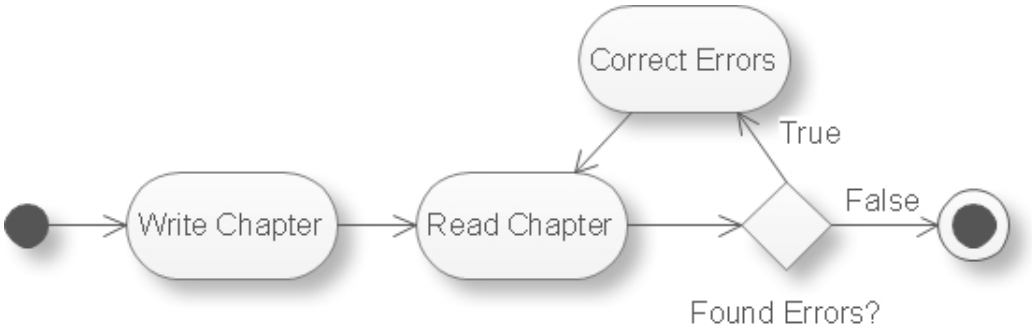
backgrounds, which makes a keyword-based query languages ideal for it. Those users who only want to quickly search for a keyword can do so with KWQL, but advanced users who want to create complex data-transforming rules can do so as well with the same query language.

Still, one question concerning query languages, or computer languages in general, is how they can be made more user-friendly.

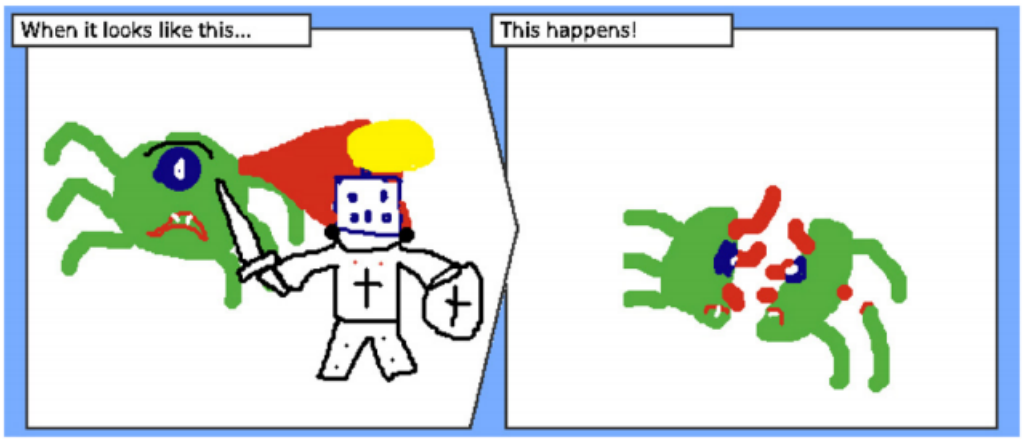
Ever since the introduction of graphic cards and computer mice, one direction that research has taken to answer this question has been the development of visual languages.



(a) QBE [3]



(b) UML [4]



(c) ComiKit [5]

Figure 1.1: Visual language examples

A wide variety of visual languages has been created over the years, including programming, modeling and query languages, which are as different in their visual formalisms as they are in their purpose or underlying textual languages.

They range from table-based database querying like the very first visual language, QBE (Fig. 1.1a), over UML (Fig. 1.1b), the modeling language which has become a fundamental part of software engineering, to such creative languages as the comic-strip based ComiKit (Fig. 1.1c) with which kids can “program” simple games.

To bring the user-friendliness of visual languages to KiWi, in the scope of this thesis a visual version of KWQL, visKWQL, together with a graphical editor, the KWQL Query Builder, have been designed and implemented.

To the best knowledge of the author, visKWQL is the first visual keyword-based query language, and had the challenge of providing features for improved user-friendliness for a textual language that was already designed for ease of use.

To accomplish this, visKWQL has been designed with an easy to understand and learn visual representation (see Fig. 1.2) that is very close to the structure of textual KWQL.

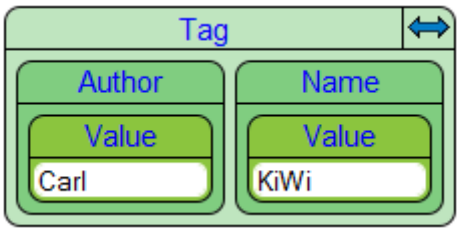


Figure 1.2: A visKWQL query

In addition to that, the KWQL Query Builder has been designed to not only allow easy and straight-forward query construction with drag & drop, but it also helps the user in different ways during query construction, aiming to allow even a new user to quickly create useful queries, without the need of reading a manual first.

This help for the user comes in different forms: The Query Builder displays tooltips explaining visKWQL elements whenever the user hovers the mouse over them, it prevents syntactical errors where possible (for example by not allowing certain drops during drag & drop) and it detects errors or problems which it can not directly prevent (for example when a query is missing an important part) and then visually displays the problematic query parts to the user, and additionally provides an explanation of why the problem occurred and how the user can solve it.

It provides additional visual features for user convenience, like coloring of elements during drag & drop to indicate if a drop action is or is not allowed in a specific location, information hiding to only display parts of larger queries the user is currently interested in, or highlighting of matching elements when the user hovers the mouse over a variable in a query.

Finally, one important feature of the system is roundtripping, which enables the user to edit either the textual or the visual query, and see any changes he makes in one representation reflected in the other. To achieve this, a KWQL parser is included in the Query Builder, to allow the translation both from visKWQL to KWQL, and from KWQL to visKWQL at any time.

1.1 Thesis Structure

This thesis is structured in the following way:

Chapter 2 introduces the area of visual languages.

After a general introduction, chapter 2.1 details their possible advantages over textual languages, and gives reasons that originally led to research into visual languages.

The following chapters introduce the different kinds of visual languages.

Chapter 2.2 gives an overview of the most commonly used visual modeling languages, like Entity-Relationship diagrams and the Unified Modeling Language, which are used to model data, algorithms, programs, business processes, and more.

Chapter 2.3 introduces a variety of visual programming languages with different visual formalisms, ranging from graphs over rules based on comic-strips, to a three dimensional approach.

Chapter 2.4 shows examples of visual query languages. Visual query languages account for the majority of visual languages, and are divided into different kinds according to the visual formalism they use.

Form-based visual query languages are introduced in chapter 2.4.1, diagram-based languages in chapter 2.4.2 and hybrid systems in chapter 2.4.3.

Chapter 3 first introduces KiWi in chapter 3.1.

An overview of the textual query language KWQL follows in chapter 3.2.

Chapter 3.3 then introduces visKWQL and the KWQL Query Builder.

The motivation and design goals are given in chapter 3.3.1, followed by a detailed look at the language and the editor features in chapter 3.3.2.

A user's manual is given in chapter 3.3.3, with an overview of the editor, an explanation of the different language elements, and a detailed walk-through of query construction with the Query Builder.

Chapter 3.4 follows with implementation details.

Finally, chapter 4 concludes this thesis with a look at some pitfalls and a summary of visKWQL.

Chapter 2

Overview of Visual Languages and Methodologies

"A picture is worth a thousand words"

This widely known quote, sometimes attributed to Napoleon Bonaparte, is testament for the preference of images over texts by many people. In some situations the quote holds true: a blueprint of a building shows more details than could be easily told with words, a photograph of a beautiful sunset might hold more than can be explained with words at all.

Of course pictures lack the syntax and semantics of text, making them inferior to written descriptions in other cases (it was John McCarthy who said, *"1001 words is worth more than a picture" [6]* to make the counterpoint to the first quote). Still, pictures are often useful to convey information, and in many areas much effort has been made to use them effectively in places that earlier belonged to written words alone.

Computer science is one such area that has changed dramatically from a text-based world to an image-based one. Where in its beginning days algorithm, data and software planning and design were mostly done in textual pseudo-code, there are now a number of structured and sophisticated visual modeling languages. Where programming was done only by entering text into a terminal, there are now a number of visual programming languages, ranging in their use of visual paradigms from employing white-spaces to convey additional syntax, to entirely forsake the use of any text and using comic panel-like images to express both syntax and semantics. And finally, where earlier only textual queries could be made to databases, today there is a huge number of visual query languages that allow the completely visual construction and editing of queries to arbitrary data sources, in addition to the visual exploration of data schemata and the browsing of database contents.

Other areas in computer science that have greatly profited from the graphical abilities of computers since the introduction of graphical display modes are graphical user interfaces and development environments and the field of program-/algorithm-visualization.

This thesis is mainly concerned with the aforementioned visual languages, for modeling, programming and querying purposes, but since the nomenclature including the term *visual* is sometimes misleading, a short comparison between the different areas employing it must be given:

Visual Languages are languages for modeling, programming, querying, or other purposes that, unlike textual languages, are *multidimensional*, meaning they employ two or

more dimensions to convey meaning [7]. They let the user manipulate program elements graphically instead of, or in addition to, specifying them textually.

Visual Programming Environments are editors for (usually textual) programming languages that allow visual programming or editing of selected parts of a program, usually the graphical user interface (GUI), by visually arranging its components. However, a programmer using them is still required to employ textual programming for the majority of operations. The most prominent of Visual Programming Environments is Microsoft's Visual Studio, that gave rise to confusion by calling its basic dialect "Visual Basic", despite it being just a textual language with the mentioned graphical GUI editor.

Software Visualization includes techniques and tools to visualize some aspect of a program, its data, code, or run-time execution. The programs are usually written in a textual language, and graphical methods are then used to gain more understanding of a program or its behavior to enable more effective use of the software [8]. An example for a software visualization tool would be a profiler that analyzes a running program and produces a bar-graph of the time the program spends in different functions, to enable the user to see where in the program code optimization would be most helpful for faster program execution. Another example would be the JHAVÉ Algorithm Visualization Environment[1], a tool to visually teach students the workings of examples like sorting and searching algorithms.

While visual programming environments and software visualization are interesting topics themselves, they are just mentioned to prevent confusion here, and the rest of this chapter will focus on visual languages.

Section 2.1 will give a more detailed introduction to the possible advantages of visual languages over textual ones, section 2.2 will introduce the most common visual modeling languages used in computer science, section 2.3 will show some visual programming languages and finally section 2.4 will provide an overview of visual query languages in more detail, since visKWQL is one such query language.

2.1 Reasons for Visual Language Research

As mentioned in this chapter's introduction, there has been considerable effort in computer science (in a survey from 1997, more then 50 visual database query languages alone are mentioned [9]) to use images instead of, or in addition to, text for modeling, programming and querying tasks. Naturally one asks, why this trend began, and what advantages visual languages have over traditional textual languages.

Mainly two things gave rise to research in visual languages: the availability of the technological means to move from text to graphics, and the trend towards improved usability.

The first modern computers had very limited capabilities. A programmer was restricted to textual input via keyboard, and textual output via terminal or printer, when developing his programs. So, naturally, programming and query languages were completely text-based.

[1] http://jhave.org/

But in the early 1980s, huge steps were made in computer hardware with the introduction of video cards and computer mice. Where earlier only linear text could be displayed, with the new hardware it was possible to use both dimensions of a monitor, since individual pixels could be addressed, made even easier by the computer mouse that allowed pointing anywhere on the screen. Furthermore, where the text was earlier always white on black, now there was support for multiple colors.

It is no surprise thus, that programmers wanted to make use of the new possibilities that allowed them to express meaning through

- Spacial placement of program elements on the two-dimensional screen
- Different shapes and sizes of program elements
- Nesting of elements
- Different colors and textures
- Icons for actions or to indicate the position of subprograms within larger programs
- Different lines and arrows to indicate relationships between elements, control flow, etc.
- Animations

and more. All this was supported by the ability to formulate and edit it easily with the computer mouse and the drag & drop techniques it allowed.

The second reason that made research into visual languages, visual query languages in particular, worthwhile, was the trend towards usability that came up roughly at the same time as the hardware that allowed visual languages.

In the early years, computers were expensive and difficult to use, and were thus only used by few people, many of which were scientists. But when hardware prices sank and the new technology was adopted in more and more places, the spectrum of the people working with computers became broader and broader. Instead of just scientists, there were now bank-workers, secretaries and others, who had to access databases for their work. In recent years, with the increasing ubiquity of the world wide web, the number of different people accessing databases has increased even more, be it the hobby-web developer using a MySQL[2] database for his blog entries, or someone interested in getting information about a book from a bibliography database stored in XML somewhere on the web.

Concerning database use, the trend went from professional users, who are experts or at least very knowledgeable in textual query languages like SQL (the Standard Query Language[3]), towards casual users, who have little or no prior knowledge of any computer language or the internal structure of a database.

Thus the goal of many visual query languages is two-fold: First, they aim to help the user understand the database they are working with. This is accomplished by supporting schema exploration and content browsing. Through schema exploration, the user can see the structure of the data in the database, visualized for instance through a diagram with circles representing objects in the database, and arrows between those circles indicating

[2]http://www.mysql.com
[3]http://www.sql.org/

relationships between those objects, in an object database. Content browsing allows the user to see the database contents without the need to formulate a query, for instance by being able to flip through pages, each displaying the values of one object in the database, with links the user can click to reach that object's children or parent element, to stay with the object database example.

The second aim of visual query languages is of course the visual formulation of queries themselves.

Some possible advantages of such a visual query formulation over the writing of a textual query are:

- Humans are visual beings and often find it easier to grasp new concepts from illustrations rather than from explanations and thus find it often easier to work with visual instead of textual representations
- Using a (perhaps even self-explanatory) visual language is often easier than learning a complex textual language with all its syntax and semantics
- Visual query languages can effectively support the user in the formulation of queries, by allowing him only to do editing operations that result in a syntactically correct query, and displaying visual cues

A more precise list of usability features of visual query formulation, or *direct manipulation visual techniques*, is given in [9]:

- Shortening of the distance between the user's mental model of reality and the representation of such reality proposed by the computer
- Reduction of the dependency on the native language of the user
- Ease in learning of the basic functionality of the interaction
- High efficiency rate obtained also by expert users, partly because of the possibility of defining new functions and features
- Significant reduction in the error rate

Of course the validity of these points changes from one visual query language to the other, and it must also be pointed out that not all visual languages reach their intended goals, but some end up introducing just an additional layer of complexity on a textual language, or even go so far in distancing themselves from any textual language that they become even more arcane to an inexperienced user than a textual language would be.

To recapitulate, interest in visual languages started with the availability of the hardware that made them possible, and with the trend towards more usability, which is also their main advantage over traditional textual languages.

Comparing and measuring things like ease of use is not an easy endeavor, and many visual programming languages are so different in their expressiveness and their intended use from textual languages, that no valid comparison can be made.

With visual query languages, however, it is a bit easier, since most visual query languages are based on textual ones and thus have the same expressiveness, as long as they

support all features of the textual version, and resulting queries can be compared easily. In [10] an attempt to prove that visual languages are easier to use than traditional ones was made:

Users with different skill levels were taught both the textual query language SQL and the visual query language QBD* [11, 12]. Afterwards, they had to construct different kinds of queries in both languages, and the authors measured the accuracy of the produced queries (how well they produced the intended result) and the time used to formulate the query. Users reached 100% accuracy with QBD* queries in most cases (while only about 90% with SQL), and query formulation in QBD* took even the expert users in some cases only about 50% of the time they needed to formulate the same query in SQL.

The authors concluded that the effectiveness of a query language varies depending on the classes of queries and the kinds of users, but the results are generally in favor of QBD*.

2.2 Visual Modeling Languages

The first kind of visual languages that shall be introduced here are visual modeling languages.

Today's world of computer science can hardly be imagined without them. They can be used to model everything from simple algorithms, to database schemata, to the behavior of reactive systems, to complex software architectures. Especially in software engineering they have become almost indispensable, with modern software systems reaching millions of lines of code, to allow engineers to keep some mental image of such a system, and to allow the planning of a complex system built from smaller modules and parts.

Since they do not depend on any computer hardware, since visual modeling can easily be done with pen and paper, unlike visual programming and query languages, visual modeling languages are neither unique to computer science nor were they first invented by it.

Flow diagrams like the Sankey diagram [13], used mostly in chemical and environmental engineering, to visualize energy, cost or material transfers between processes, have existed for more than a century. There is also hardly anyone who hasn't used, or at least seen, some kind of Flowchart [14], which are still used in computer science. But they are so easy to understand that they can be used to visualize nearly any arbitrary process, from the workings of a machine in industrial engineering, to the humorous illustration of how to choose the right breakfast ingredients.

And, especially in recent years, more and more enterprises try to improve process efficiency and quality by employing Business Process Modeling [15], which in the past also made use of flow diagrams and flowcharts, among other visual modeling languages like Gantt charts, PERT diagrams and others, and for which today exist a number of specialized modeling languages like UML, which will be introduced in chapter 2.2.3.

Returning to visual modeling languages in the realm of computer science, it also needs to be mentioned that lately the distinction between visual modeling and visual programming languages has become blurred in some cases. For some modeling languages like UML, there now exist sophisticated tools that can both generate UML diagrams from source code, and source code from UML diagrams, which makes it possible in a way to do some programming with the modeling language.

In the following sections, three important visual modeling languages will be introduced: Structograms were the first adaption of flowchart techniques to structured programming; Entity-Relationship diagrams are the most widely known tool to model databases; and UML is the most widely and best known modeling language for a variety of applications from object-oriented programming to business process modeling.

2.2.1 Structograms

Structograms, or Nassi-Shneiderman diagrams, named after their inventors, where introduced 1973 in [16], as a method to model computation in the simply ordered structures of structured programming.

Earlier, flowcharts were used to model algorithms or programs. But they lack a direct representation of some of the more restrictive control structures of programming languages, like iteration, but on the other hand support the now unwanted notion of the unrestricted go-to, which made the translation of flowcharts to programs a one-to-many relationship, where not all possible outputs were optimal.

In addition to the (at the time, new) ideas of go-to-less and structured programming (where programs consist of a row of smaller parts, and where no unrestricted go-to is allowed), the second new idea, that inspired Nassi-Shneiderman diagrams, was that of Top-Down programming, the technique of analyzing an idea (which may be a program, a subroutine, or any other modular part of a program) to form simpler ideas, and recursively apply the technique.

Structograms consist of four basic types of building blocks: process, iteration and decision functions, and an additional BEGIN-END symbol for languages like ALGOL.

The commonly used items, supported by today's software with support for Nassi-Shneiderman diagrams are:

Instruction represented by a simple box containing the instruction text

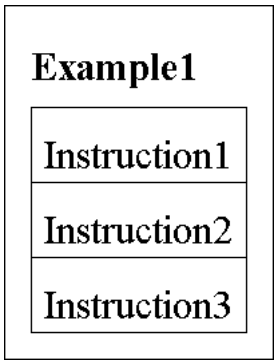
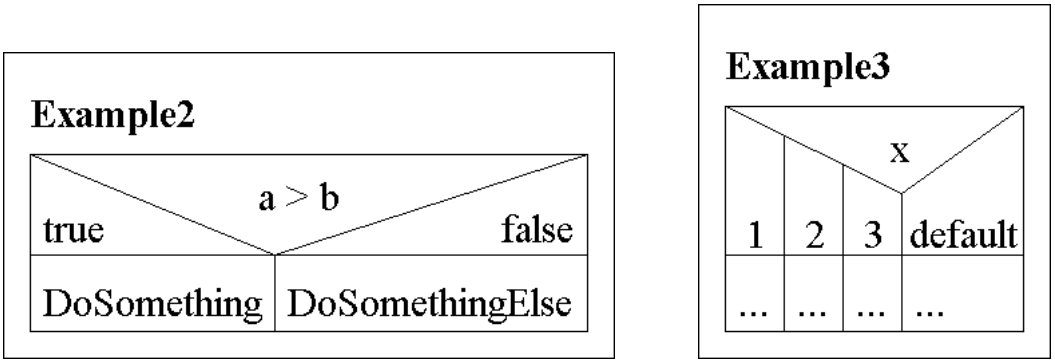


Figure 2.1: Simple Instructions

If-then-else represented by a box containing a triangle with the if-clause, which splits the following item into two halves, one containing the then-clause, the other the else-clause

Case represented by a box with a triangle like the if-then-else, but splitting the following item into multiple parts, one for each case plus the default action

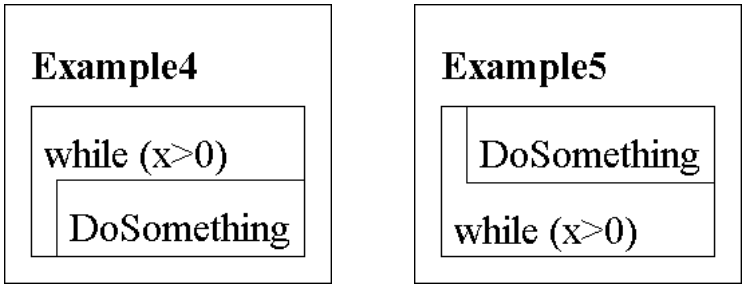


(a) If-Then-Else (b) Case

Figure 2.2: Decision Functions

While represented by a box with the boolean condition, that encompasses the instructions to be executed while the condition is true

Repeat represented like While, only that the instructions come before the condition. This corresponds to the do-while loop in languages like C



(a) While (b) Repeat

Figure 2.3: Iterations

Figure 2.4 shows a complete factorial algorithm modeled as a structogram. First the input n is checked in an IF-THEN-ELSE clause, returning an error if it is smaller than zero, since the factorial is undefined for negative numbers. It then initializes variables result and i, and computes the factorial with a while loop.

The structogram's intended advantages over traditional flowcharts are:

- The scope of iteration is well-defined and visible
- The scope of IF-THEN-ELSE clauses is well-defined and visible; moreover, the conditions on process boxes embedded within compound conditionals can be easily seen from the diagram
- The scope of local and global variables is immediately obvious

- Arbitrary transfers of control are possible
- Complete thought structures can and should fit on no more than one page
- Recursion has trivial representation

To be able to more precisely model programs in modern programming languages, some tools like Structorizer[4](which was used to create the images in this section) have introduced additional elements like for-loops, CALL and JUMP.

The authors of the original paper on the other hand saw some of the things, which were seen as shortcomings by others, as a virtue: the absence of branch representation forces the programmer to design programs in a structured manner, and to not develop the habit of using go-tos which make code harder to read and debug. And since no more than a handful symbols can be drawn on a sheet of paper, the programmer is forced to modularize his programs.

One more quality of structograms that can be seen from two sides is their similarity to program code. This makes it possible to very easily translate a diagram to code, but on the other hand raises the question why one should draw the diagrams at all and not just model with pseudo-code.

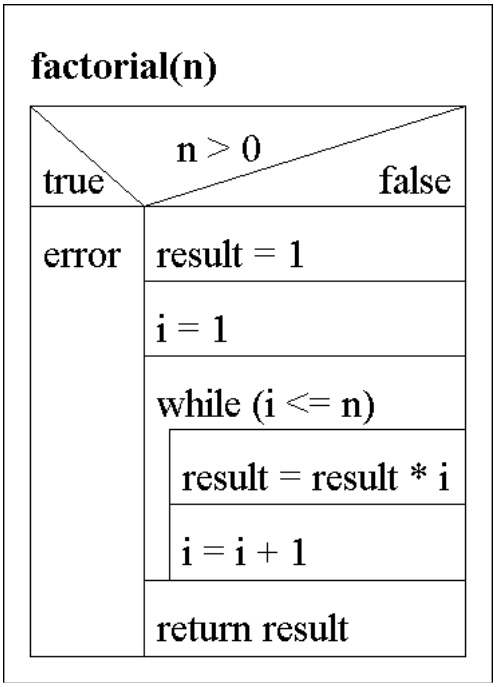


Figure 2.4: An example of a factorial function as a structogram

2.2.2 ER Diagrams

Next to algorithm or program modeling, database modeling is an important area that has greatly profited from visual modeling languages.

One of the most popular high-level conceptual data models, especially for relational databases, is the Entity-Relationship (ER) model, introduced in 1976 in [17], together with a diagrammatic notation for it, called Entity-Relationship diagrams.

[4]http://structorizer.fisch.lu/

The ER model describes data as entities, relationships and attributes, or sets thereof.

Entities are "things" of the real world, either with a physical existence (for example a person) or a conceptual existence (for example a university course)

Attributes are the particular properties of an entity, used to describe it (for example the name of a person)

Relationships are associations between entities (for example *attends* could be a relationship between a person and an university course)

Attributes can either be atomic or composites. For example, an address, consisting of a street name and number, would be a composite attribute, while the street name alone would be an atomic attribute. They can be single-valued (for example the age of a person) or multivalued (for example all the colors of a car). They can be stored (directly assigned a value) or derived (computed from other information, for example when a person has the birth date as an attribute, the person's age could be computed from that date and the current date as a derived attribute) and they can be null values to indicate absence of that attribute. [18]

Each entity also has one or more key attributes, which are unique to each instance of the entity and allow the entity's identification.

Relationships between entities have a degree, indicating the number of participating entity types. Relationships of degree two and three are called binary and ternary relationships, respectively. They may also have attributes similar to those of entities, and structural constraints for the minimum number of participating entity instances, and for binary relationships also for their maximum number.

Entities in a relationship may be assigned role names, indicating the role they play in that particular relationship. They may also participate multiple times in a relationship and hold different roles, allowing for recursive relationships.

Entity Relationship diagrams allow the visualization of these concepts:

Entities are represented by a square containing the name of the entity

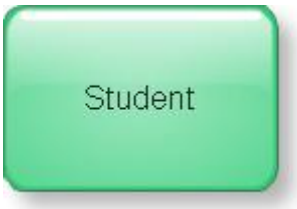


Figure 2.5: An Entity

Relationships are represented by diamond shapes, containing the name of the relationships, and connected with lines to the entities participating in it

Figure 2.6: A relationship between two entities

Attributes are represented as ellipses connected to an entity, a relationship or another attribute in the case of composite attributes. Key attributes are underlined

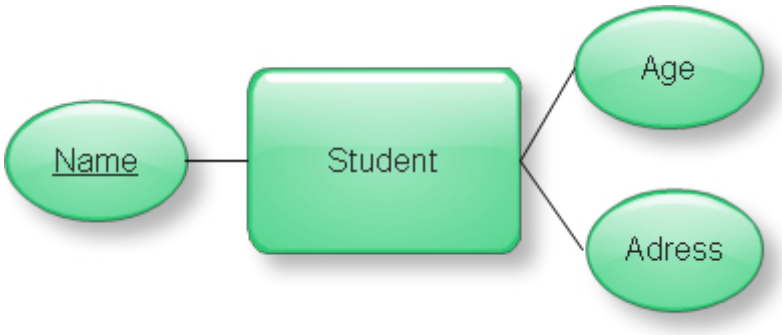


Figure 2.7: An entity with attributes

Role names and constraints are written on the lines between entities and relationships

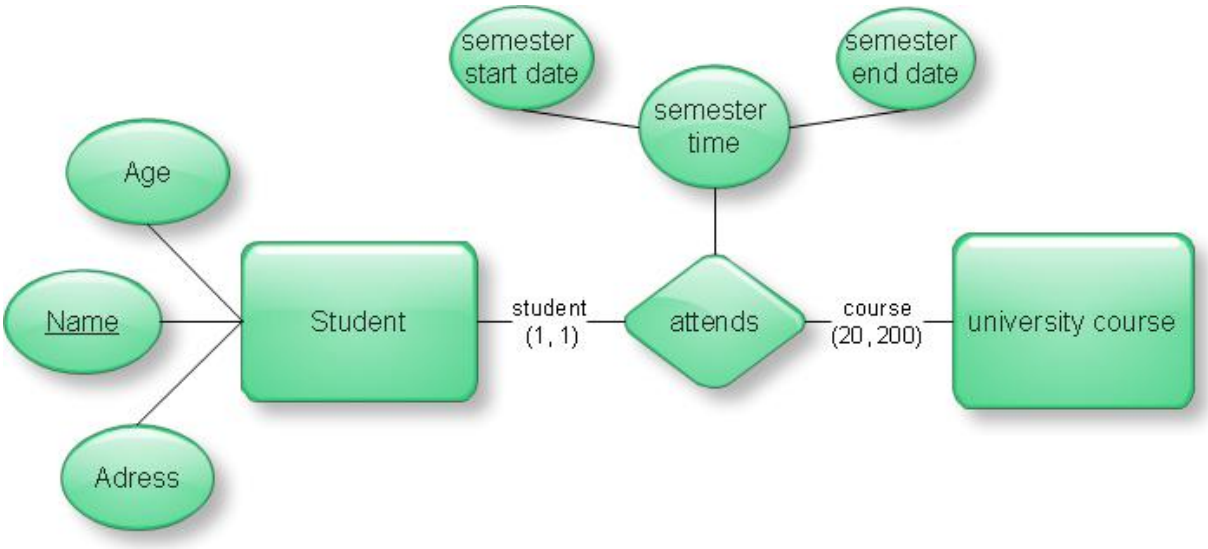


Figure 2.8: A simple ER diagram for students attending a university course

Figure 2.8 shows a very simple ER diagram: students, each with a name, age, and an address (with name being the key attribute[5]) attend a university course. They only attend

[5]which is of course not optimal, since two students can have equal names, but just used for simplicity here

the course during a semester, indicated by the composite attribute *semester time*. Below the student's and university course's role names are numbers for structural constraints: the (1, 1) below student indicates that both the minimum and maximum number of university courses one student can attend is one. He cannot attend multiple courses, and when he attends zero courses, he is not a student. The constraint of (20, 200) of the course indicates that each course must have between 20 and 200 students attending it. In this example, only 200 students would fit into the lecture hall, and if there were less than 20 students attending, the course would be canceled.

It must be added here, that in the 30 years since Chen's original paper (which introduced the above visual representations), the initial representation of ER diagrams has changed considerably. The most common changes tried to make the diagrams more compact, with the attribute names appended as a list below the entity name within the entity's square, with different symbols at the beginning or end of connection lines to indicate certain constraint values, and with the relationship shape omitted and the relationship name added to the connection line. One of the modern conventions for ER diagrams is to model them like UML class diagrams.

2.2.3 UML

UML, the Unified Modeling Language [4], is a standardized, general-purpose visual modeling language for software engineering tasks.

It succeeds a number of concepts and techniques, like object oriented software engineering, and aims to be a single modeling language for all processes of software engineering, by combining data-, business-, object- and component modeling, supported by extensibility and an underlying meta-model.

UML, in its 2.0 specification, is based on four parts:

- The Infrastructure, which defines the meta model
- The Superstructure, which defines notation and semantics for all diagrams, based on the Infrastructure
- The Object Constraint Language (OCL), which is used to define rules for model elements
- And the UML Diagram Interchange, a definition of how to exchange UML diagram layouts

One has to note that OCL is used to define rules for model elements, and not diagram elements. Each UML diagram is only a partial visualization of a system's model, and usually a multitude of diagrams, both static and dynamic, to describe both structure and behavior, together with semantic additions, are used together, to model a complete system.

In the years since the first UML draft in 1997, the language has become widely accepted in the software engineering world, and today there is a large number of tools to support it. As mentioned earlier, this is where UML could be said to transcend the realms of modeling

and programming, since for example UML class diagrams can be directly translated into object oriented programming languages, or created from code written in those languages, by a number of tools. UML also allows the addition of notes to all diagram elements, which can be used to specify program code for methods within a class diagram, so that it is possible, with the right tool, to generate simple, but complete, working programs entirely from one UML class diagram.

The versatility of UML however lies in its inclusion of multiple diagram types, through which the different aspects of a software system can be modeled. As of the time of writing, the UML standard includes 13 different types, grouped into structure, behavior and interaction diagrams:

Structure diagrams define the elements a system consists of:

Class diagrams are probably the most widely known variant of UML diagrams. Classes (of object oriented programming languages) or interfaces are represented by rectangles with the class (or interface) name and a list of its attributes and methods, together with indicators whether those are public, protected or private. Lines between rectangles indicate associations between classes (or interfaces). The end-points of the lines can be hollow or filled diamonds, to indicate aggregation or composition. Properties like role names and multiplicity can be added to those connections. Different arrows are used to indicate class relationships like generalization and dependency. Class diagrams are also used to model data with the entity-relationship model. See Figure 2.9 for an example of a class diagram[6].

Component diagrams are used to describe how components (modular, encapsulated parts of a system, defined by interfaces they provide and interfaces they require) form larger components or the whole system. Components are drawn as rectangles, with so-called assembly connectors connecting the provided and required interfaces of different components.

Composite structure diagrams are similar to component diagrams, but describe the internal structure of classes, and the connections of interacting parts within them.

Deployment diagrams can model the relationship between a system's software and hardware. The diagrams consist of *nodes*, representing hardware, for example a database server, and *artifacts*, representing software parts, for example a database table. Rectangles of artifacts are drawn within rectangles of the nodes those artifacts will physically exist on in the running system. Lines can connect physically connected nodes in the diagram.

Object diagrams show the structure of a system at a specific time. They do not represent object classes, like UML class diagrams, but instances of those classes during run-time of the system, together with specific values of their variables at a specific point in time. Object diagrams also usually don't show the whole class hierarchy, but only classes and attributes of particular interest for an example or a test-case.

Package diagrams are used to model the structure of packages (groupings of classes, components, use cases, etc.) and their dependencies. Lists of elements are writ-

[6]Image taken from http://en.wikipedia.org/wiki/Composite_pattern

ten within rectangles representing packages, with arrows between packages representing package import or merge.

Behavior diagrams define the behavior of the elements a system consists of:

Activity diagrams can be used to model the overall flow of control in a system, from the perspective of business or operational work-flow. The diagrams consist of a start- and end node, activity nodes, arrows between those nodes to indicate control flow, and special action nodes to model choice, iteration and concurrency. See Figure 2.10 for an example of a very simple activity diagram.

State diagrams are used to model stateful systems: the diagrams consist of circles to indicate start- and (optional) final state, rounded rectangles denoting states (and optionally activities that are executed in those states), special symbols for fork and join, and arrows between states representing state transitions, together with optional event names, guards and actions for transitions.

Use case diagrams model a system as it interacts with the outside world. The diagrams consist of symbols for different actors (persons or machines interacting with the system, for example an user or an administrator), use-cases (specific behavior of the system in response to specific actions or input from actors) and relationships between those use-cases (inclusion, extension and generalization).

Interaction diagrams define the interactions between the elements a system consists of:

Communication diagrams model interactions between system elements via messages. Objects or other system parts are represented by squares connected by lines. Each line represents one message (or method invocation) between those objects, annotated with a chronological number.

Interaction overview diagrams are high-level activity diagrams, with each node in the diagram representing an entire interaction diagram

Sequence diagrams show how different co-existing objects or processes communicate with each other. They are represented by vertical lines, with horizontal arrows between them indicating messages or method invocations. Where an object receives a message an *activation box* is drawn, indicating that in response to the message the object is actively doing something. During the activation period, the object itself may send messages to other objects. This way, a time flow from top to bottom is established to easily see the chronological order of messages. See Figure 2.11 for an example[7].

Timing diagrams are sequence diagrams with reversed axes, so that time flows from left to right, with specific timing constraints added to the diagram.

While all UML diagrams are applicable to software systems, many are not restricted to them, but can be used to model many types of systems or activities. Use case diagrams for example can be used to model the expected interactions between people in a business, where use-cases are not executed by a machine but by persons. Activity diagrams can not only be used to model control flow in a software system, but also to model the process of decision

[7]Image taken from http://en.wikipedia.org/wiki/Sequence_diagram

making within a company's management, or the process of preparing a dish. The same is true for state diagrams and others.

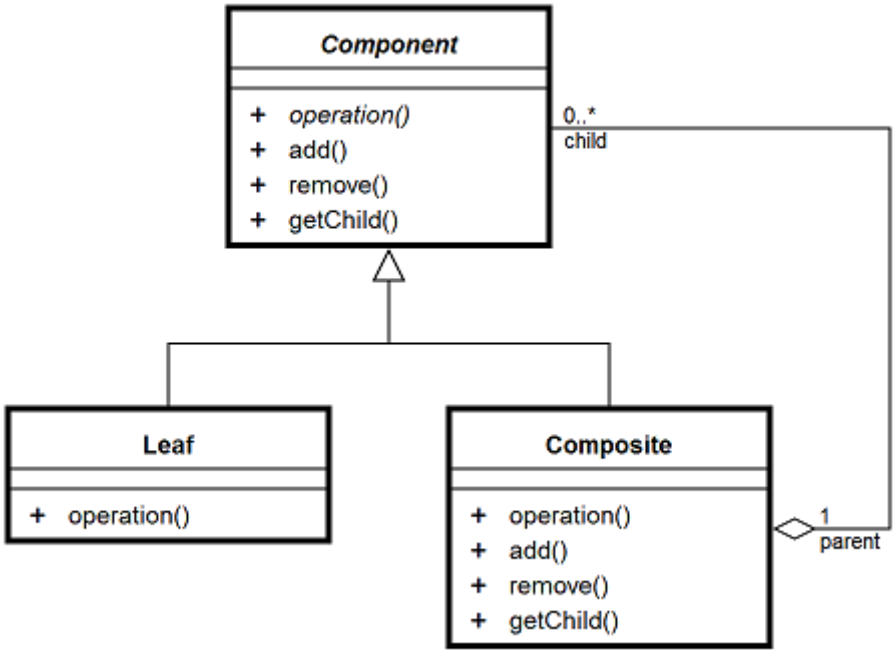


Figure 2.9: An UML class diagram of the composite design pattern

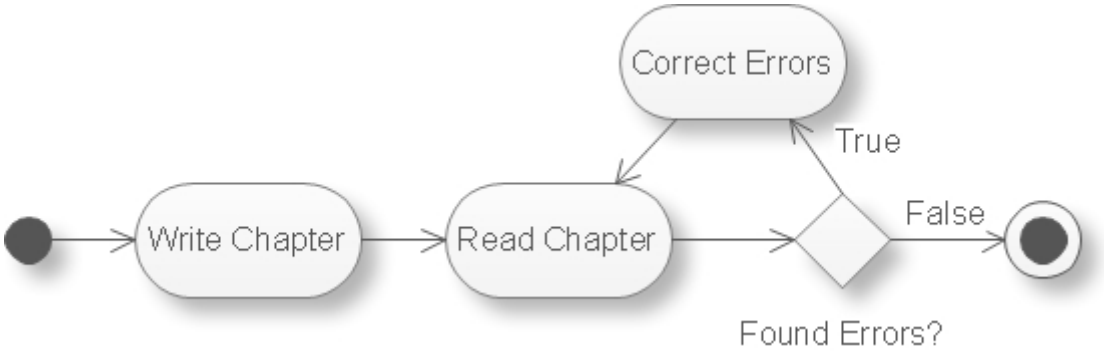


Figure 2.10: A simple UML activity diagram for writing a chapter in a thesis

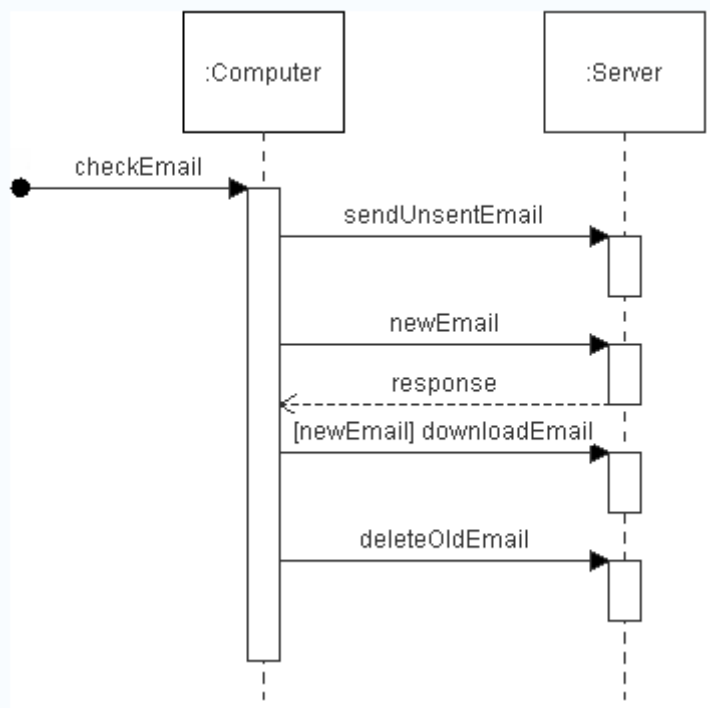


Figure 2.11: An UML sequence diagram for the interaction between an email-client and server

2.3 Visual Programming Languages

While visual modeling languages, especially UML, have become a solid pillar of software engineering, visual programming languages have not been able to become as important or widely used.

Their main problem is scale: while visual modeling languages are used to get a higher level view of a system, and reduce detail, programming languages must specify the details of a system. And with modern software ranging anywhere from some thousand to a couple million lines of code, the dilemma visual programming languages face is easy to see: representing everything in two or more dimensions, the amount of space that larger programs occupy quickly gets out of hand.

Textual languages are also easier to read and debug by humans, since text flow goes from left to right and top to bottom. For visual programming languages however there is no such constant flow direction, and where in the 2d-space (or 3d-space in some cases) a certain object, algorithm or instruction lies is entirely up to some conventions of the specific language, and the whim of the programmer.

This has lead to a low adoption of visual programming languages in most areas where programs consist of more than a handful lines of code. However, in some areas they have gained some more recognition.

In the field of data flow programming they have advantages over textual languages since diagrammatic languages are able to directly visualize the data flow, unlike textual languages [19].

They are also suited well for simple, high-level programming tasks, where they come in the form of *animation-* or *simulation-makers* (see for example ARK [20], where the pro-

grammer can create simulations, or "alternate realities", from high level building blocks representing for example planets and physical laws like gravity).

A third field that profits from visual programming languages is education.

The aforementioned *animation-* or *simulation-makers* also reach into education sometimes, mostly for younger children, with the whole system being game-like (for example Kid Sim [21], where children can let sprites on the screen interact with each other with programming-by-example techniques and by creating simple graphical rewrite rules. For example, a rule with a sprite of a gorilla left to the image of a rock in the first part of the rule, and the the sprite of the gorilla to the right of the rock in the second part of the rule, would let the gorilla "jump" over the rock when he walks from left to right in the simulation and encounters a rock).

But visual programming languages are also useful to teach programming to people new to it, since the user might be able to better grasp the concepts of things like objects when they are visualized to him in some way, and since visual programming languages can better enforce program correctness, by allowing only certain actions, to help the user create working programs.

An example of an educational visual language will be shown in chapter 2.3.1.

Another, newer, field where visual languages are applicable, are transformation languages for semi-structured data, where programs are relatively short and visual programs can fit on one screen.

To compare and evaluate visual programming languages, a classification scheme is necessary. Through their defining characteristics, they can be grouped into five different types [22]:

Purely visual languages rely on visual techniques throughout the whole programming process. The programmer manipulates graphical representations to create a program, which is then executed and debugged in the same visual environment. The compilation of the program is done directly from graphical entities, with no underlying or intermediate textual language.

Hybrid textual and visual systems combine visual and textual elements. Programs are created visually and then translated into a textual language, or written in specialized systems which allow interspersing textual code with graphical diagrams.

Programming-by-example systems allow the programmer to create and manipulate graphical objects to teach the system how to perform tasks.

Constraint-oriented systems are the earlier mentioned *simulation makers*, where the programmer models physical objects by creating graphical representations and adds constraints to them, to model physical laws acting on the objects.

Form-based systems borrow their visualization and programming metaphors from spreadsheets, and programming is done by altering groups of connected cells over time. Program execution is equally represented by cells altering their states.

In the following sections, some notable visual languages with different purposes and visual concepts are introduced.

2.3.1 SIVIL

SIVIL (SImple VIsual Language) [23] is a declarative visual programming language designed to teach complete novices (who are new to programming concepts such as loops) the basics of programming after the functional programming paradigm.

The philosophy behind SIVIL is that textual programs suffer from clutter, and detail should be put in the background or hidden altogether.

Thus SIVIL uses icons and pictures to represent loops, variables, objects, and all its other structures, and lets the user connect them to form programs. It also introduces the concept of *enclosers*, which are pieces of a program, for example one instruction or one algorithm, that can be "closed", to be represented by a single square with a name in the overall program, to reduce detail and make it easier for the student to see the overall structure of, and data flow in, his program.

Figure 2.12 shows part of a screen-shot of a SIVIL program that takes the square root of a constant and prints it. The left part of the image shows the overall program, with icons representing a constant and a print action, and the encloser "square root" between them. The right part of the image shows the opened encloser, with icons representing the entry and exit points, and a textual instructions to compute the square root of the input value.

This co-existence of graphical representations and textual instructions make it easy to classify SIVIL as a hybrid text and visual system according to the classification system mentioned previously.

SIVIL allows using a "skin" for its variables, enclosers and primitives, allowing for example a teacher to create a set of symbols that are especially meaningful to his students, to represent the different structures of SIVIL, or to serve as a form of additional commenting.

The use of comments, or more precisely, teaching good documentation skills, is another goal of SIVIL, which it aims to reach by allowing the commenting of whole programs as well as all enclosers with both short descriptions and detailed information boxes.

SIVIL also has special support for debugging, by letting the user speed up or slow down program execution to view the program as it runs, and letting him inspect the values of variables as they are received by enclosers.

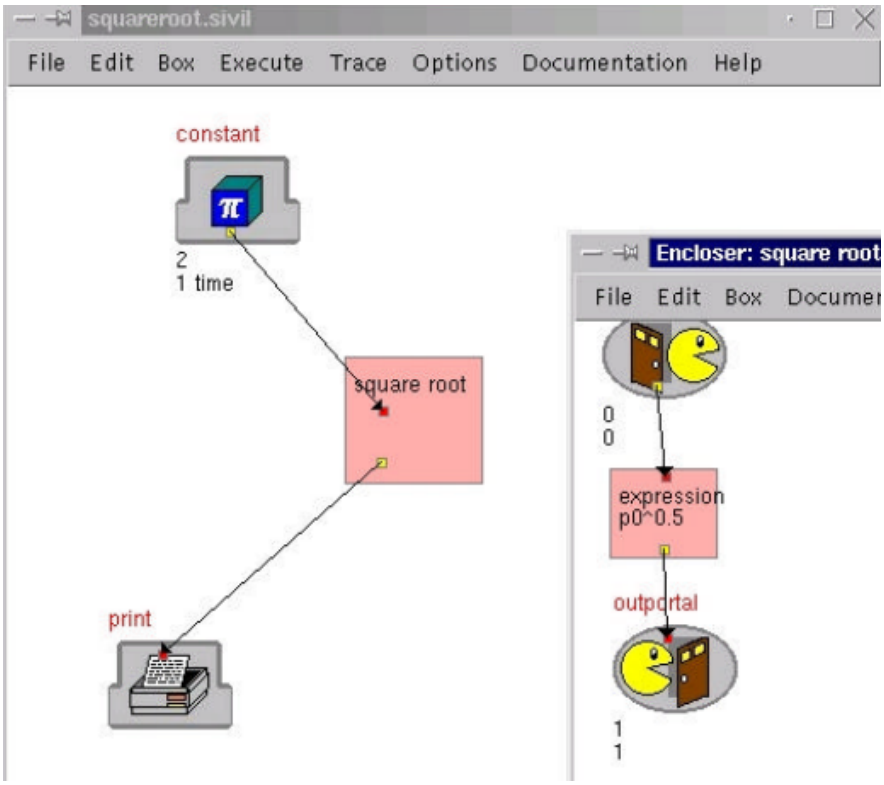


Figure 2.12: A simple SIVIL program

2.3.2 VXT

VXT [24] was the first visual XML transformation language.

Its aim is to provide a language and interactive environment specifically tailored for programming XML transformations in a visual manner, to save the programmer from having to maintain mental models of the data structures he is working with.

VXT is based conceptually on the declarative textual XML transformation language XSLT [25], based on source-driven transformations consisting of rules that can be cascaded, that is applied one after the other during a recursive descent of the XML document tree.

The programs, or transformations, can be run either from within the visual environment, which allows for easy debugging, or can be exported as either XSLT or Circus[8] code.

XML transformation languages usually present the user with three different abstractions: the XML documents to be transformed, document type definitions (DTDs), defining the structure of the XML documents, and the transformation rules. In textual languages, these three usually look quite different and have different syntax and semantics from one another.

VXT aims to make the transformation programming easier for the user, by not only representing all of them visually, but by using a single, tree based, visual representation for all of them.

A problem with visual representations of XML tree structures lies in the fact, that, with growing document size, the trees get extensively wide or high, making them unable to fit

[8] a textual XML transformation language also introduced in [24]

on one screen and requiring the user to scroll around a lot, which makes finding nodes and editing the trees a chore.

VXT tries to solve this problem by representing an XML tree with a variation of a *treemap*, where children are not connected to their parent nodes with lines or arrows, but nested inside of them, similar to what we will later see in visKWQL.

The environment allows zooming in and out of those treemaps, allowing the user to easily see both the overall structure and explore the details.

In VXT, child nodes are arranged in one line, following a horizontal flow, within their parents. Attributes of XML elements are not considered as true children, and therefore not nested, but laid out above the bottom edge of the element they belong to.

See figure 2.14 for an example of VXT's representation of XML documents. Each XML element is represented as a rectangle or square, with its name above it, and the rectangles representing its children within it. The only attribute in the example is "ref", which can be found above the bottom line of the "mail" rectangle.

An element's type is represented by its shape and hue, a gray diamond representing text data in the example.

```
<mail ref="ep47">
    <date>26-01-2001</date>
    <recipient>vion-dury@xrce.xerox.com</recipient>
    <sender>pietriga@xrce.xerox.com</sender>
    <subject>Jazz</subject>
    <textbody>
        <p>Hello Jean-Yves,</p>
        <p>
            Have you listened to
            <cite>My Favorite Things</cite>
            by John Coltrane?
        </p>
        <p>Emmanuel</p>
    </textbody>
</mail>
```

Figure 2.13: XML document for a mail

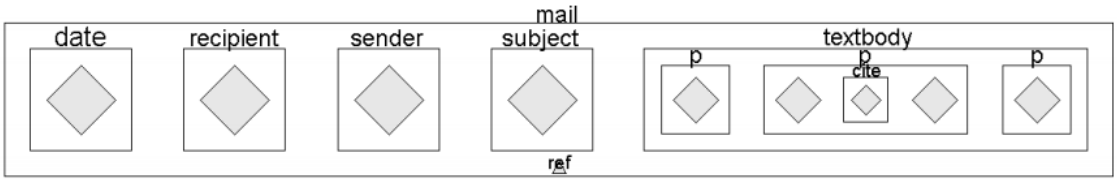


Figure 2.14: VXT's treemap representation of the XML document

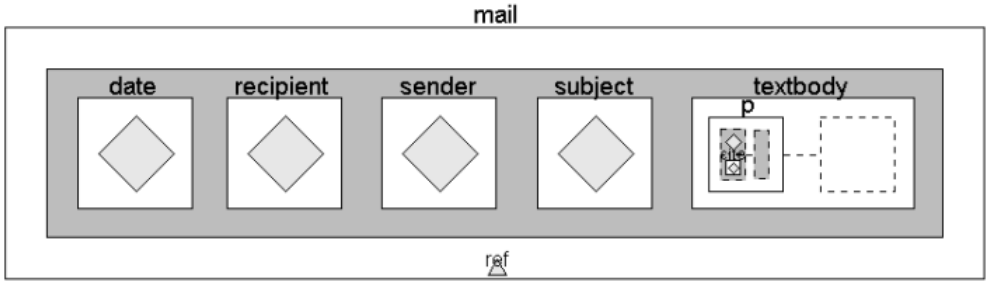
As mentioned before, VXT tries to represent XML documents, DTDs and transformation rules in the same way, despite their differences.

DTDs can specify sequence, choice and cardinality. Sequence of elements is represented in VXT by nesting the nodes in an extra blue rectangle. Choice between elements is represented by coloring them green and stacking them on top of each other. To indicate cardinality, an element's outline is dashed to indicate "zero or one", an additional, empty,

square with dashed outline is appended to represent “zero or more”, and the same square is appended, while the outline of the original element stays solid, to represent “one or more”.

```
<!ELEMENT    mail        (date,recipient,sender,
                          subject,textbody)>
<!ELEMENT    date        (#PCDATA)>
<!ELEMENT    recipient   (#PCDATA)>
<!ELEMENT    sender      (#PCDATA)>
<!ELEMENT    subject     (#PCDATA)>
<!ELEMENT    textbody    (p)+>
<!ELEMENT    p           (#PCDATA | cite)*>
<!ELEMENT    cite        (#PCDATA)>
<!ATTLIST    mail  ref  CDATA  #REQUIRED>
```

(a)



(b)

Figure 2.15: The mail document’s DTD and VXT’s representation thereof

Figure 2.15 shows the DTD of the XML document from figure 2.13, together with its treemap representation.

Transformation rules consist of three conceptual parts: Node selection, data extraction, and data reorganization. The former two are merged into *Visual Pattern-Matching Expressions* (VPMEs), which can be considered as filters, applied to a source structure, to extract some information from it.

VPMEs thus have the ability to specify nodes to be extracted or not to be extracted, selection constraints (nodes that must exist, must not exist, or may exist), and nodes which are direct parents/children or descendants/ancestors of a given node. Each of those expressions is represented by a different coloring of the nodes and their borders.

For the third part of transformation rules, the construction of a new document, the user creates a new treemap from the elements he marked as “to be extracted” and possibly new elements. He can then connect nodes in the source and the new treemap. Three possible connections, identified by a symbol in the middle of the line, are possible: deep copy of a node (xsl:copy in XSLT), text extraction from a node (xsl:value-of), and rule application to a node (xsl:apply-templates).

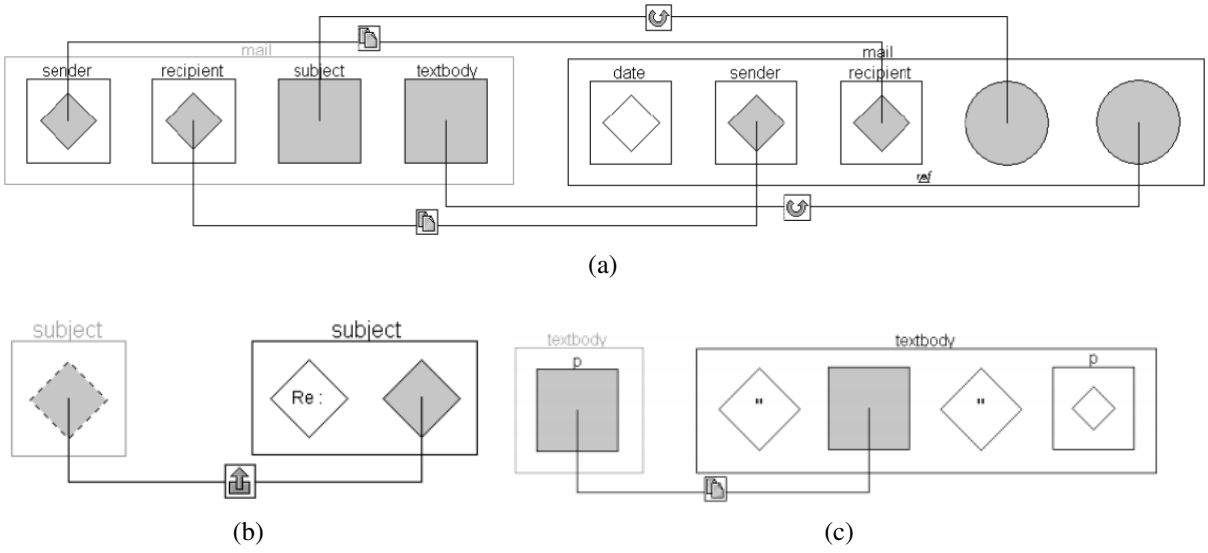


Figure 2.16: A complete VXT program

Figure 2.16 illustrates a complete VXT program, to create a reply template to a mail, conforming with the schema in figure 2.15.

In the first rule (figure 2.16a), a new document is created. On the left side of the rule, a "mail" element is selected, which must contain sender, recipient, subject, and a text body. Sender and recipient must be non-empty (indicated by the filled diamond), while subject and text body are only required to exist, but may be empty.

When the rule is executed, a new "mail" element is created, together with date, sender, and recipient elements. The source mail's sender is copied to the new mail's recipient, and the recipient is copied as the new sender. Subject and text body are not directly copied to the new mail, but other rules are first applied to them (indicated by the rounded arrow symbol)[9].

The rule in figure 2.16b adds the string "Re:" to the mail's subject line. It creates a new "subject" node, inserts a text node containing the text "Re:", and appends a copy of the text value of the source subject.

The last rule (figure 2.16c) transforms the text body of the mail. It creates a new "text-body" element, copies the source's text (a set of "p" elements) into it, surrounded by quotes, and adds a new, empty paragraph element, for the user to write in.

VXT also allows the representation of iteration and conditional constructs (xsl:for-each, xsl:if, xsl:choose, and xsl:otherwise) via different shapes and colors, in addition to the naming of rules, to reach a high expressive power, similar to XSLT.

Furthermore, the system helps the user to avoid syntactic and certain semantic errors, due to its nature and by constraining VPME and result-fragment construction.

[9]note that VXT draws those elements as a circle in the construct part, indicating that their type there is unknown, since they are subject to other transformation rules before being included there

2.3.3 Yahoo! Pipes

Yahoo! Pipes[10] [26, 27] is a visual programming language and environment that allows the creation of RSS-feeds[11] by remixing other feeds, and the creation of simple web-applications as data mashups[12] from different web-pages, - feeds and -services in an imperative manner.

The name "Pipes" is taken from Unix pipes, which allow the chaining of command-line tools.

Similar to Unix pipes, Yahoo! Pipes lets the user create programs by connecting pre-configured modules together, so that one module's output becomes another one's input.

Available modules include user input fields, a "Fetch" module that retrieves a RSS-feed URL, a "Sort" module that can, for example, alphabetically order the entries in a RSS-feed, modules for arithmetic computations, a rich number of string manipulation modules, including an URL builder, and many more. Most modules have a number of parameters, which can be set by the programmer or other modules.

One feature that Yahoo! Pipes have in common with the KWQL Query Builder is, that it handles web data and the programs can be constructed in any web browser, without the need to install any software.

Programs are constructed in an easy and straight-forward manner. Modules, represented as boxes, can simply be selected from a menu to be added to the workarea, where the user can fill in their parameters. Each module has one or more "terminals", represented by small circles. Most modules have one output terminal, at the bottom of the module's box, and a number of input terminals, which usually are placed next to a parameter field that will receive its value from it.

The user can then simply wire the different module's terminals together to enable data flow and so create a program.

The module boxes include some icons in the upper right corner that correspond to those of windows in operating systems, with a question-mark icon that will open a field to show a description of the module, an icon to "minimize" the module's box and hide its contents, and an "X" icon to delete the module.

[10] The system is available online at http://pipes.yahoo.com/pipes/

[11] "Really Simple Syndication": a format containing text and metadata, used to publish website content updates

[12] aggregation of data (or functionality) from different sources

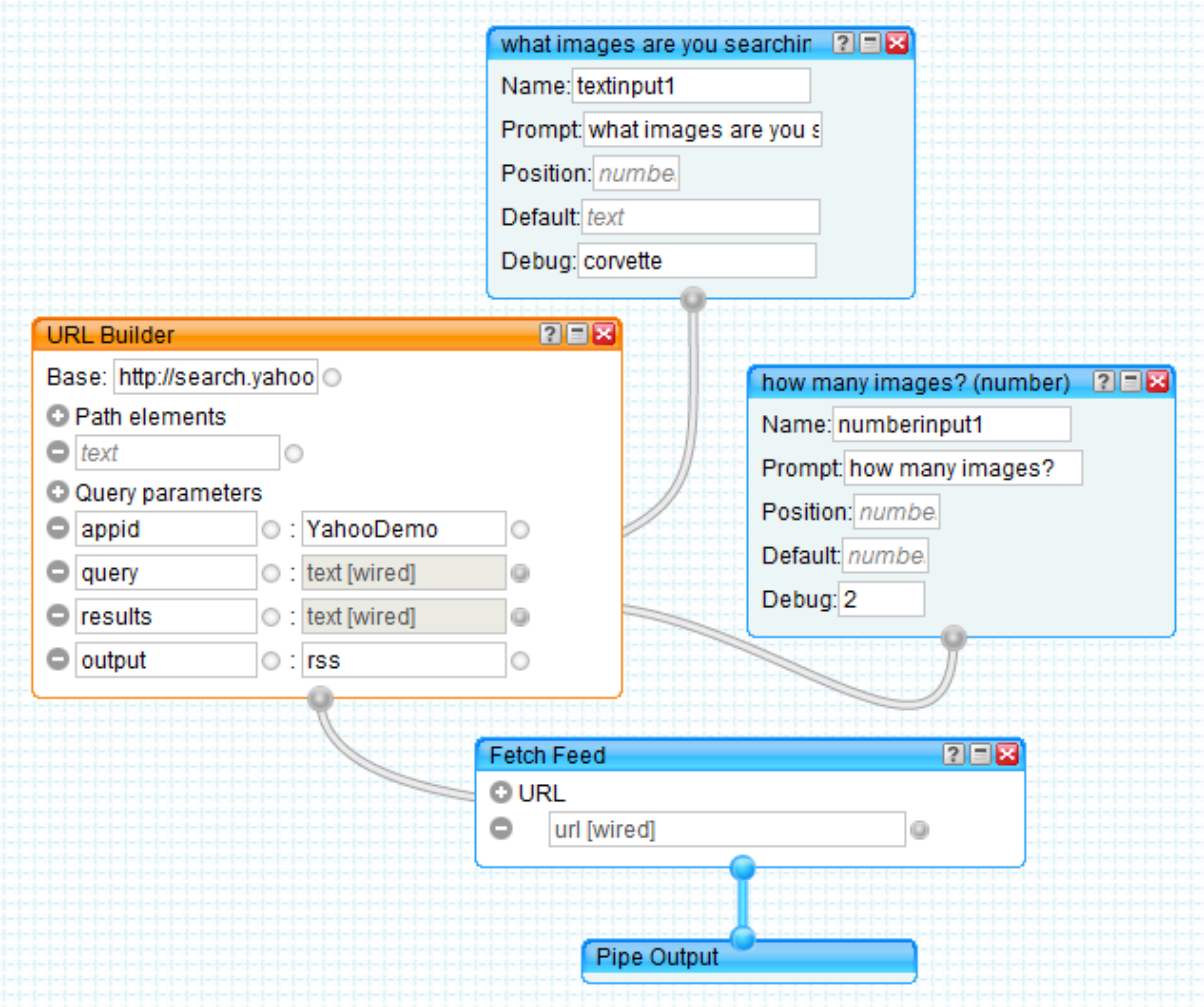


Figure 2.17: A Yahoo! Pipes application to search for images

Figure 2.17 shows an example of a Pipes program for image searching.

The upper and right boxes are user input modules, which will show input text fields to the user when he executes the program, and ask him, what images he is searching for, and how many images he wants to get returned. The information from these fields is then channeled into an URL Builder module (which is orange in the image, because it was selected with a mouseclick), which takes those parameters, in addition to some parameters that the programmer entered, like the search machine's base URL, and constructs an URL from them that will search the specified images. Finally, this URL is fed into a Fetch Feed module, which will retrieve this URL and output the data it finds there, which will be an image search result page.

While Yahoo! Pipes allows some high-level actions like fetching a feed in a very simple manner, its expressiveness is lacking in some aspects, especially concerning low-level actions. So, for example, it is not possible to add some kind of loop within a module, requiring the user to specify a very large number of different parameters by hand in some cases. Another disadvantage of the system is that the created pipes require the Yahoo! server to be executed, and can thus not be used as stand-alone programs.

This has led to a relatively low success of Yahoo! Pipes, although it fared better than

projects with similar aim, like Microsoft Popfly or Google Mashups Editor, which were both discontinued.

2.3.4 JBoss jBPM

JBoss jBPM (java Business Process Management Engine) [28] is a system that, unlike a number of visual programming language systems that are no more than academic prototypes, gets used today in the development of web applications like KiWi. It is a tool to visually program business processes and user interactions within the JBoss Seam Framework[13] with the use of graphs, that are defined in an XML dialect called jPDL, which can be visually edited through a plugin in the popular Eclipse Development Environment[14].

jBPM allows the visual programming of pageflows and business processes.

A jPDL process definition defines the pageflow (or task; a path through different connected web pages) for a single conversation (a short-running interaction with a single user), and can be used for example to define what pages a user encounters when he browses and orders items in an online shop.

The difference between a jPDL pageflow and a pageflow modeled only through navigation rules is that a jPDL pageflow is stateful. For an informational website, free-form navigation is unproblematic, web applications, however, profit greatly from statefulness, to restrict the user's interaction in a meaningful way, for example to not allow a user of an online shop to reach the "confirm order" page unless he has entered a shipping address into the system on another page.

A business process is a long-running process that may span multiple interactions with multiple users, and thus needs to deal with multiple concurrent paths of execution. More formally, a jBPM business process is a well defined set of tasks, together with well defined rules about who can perform a task, and when a task should be performed. The state of a business process in jBPM is automatically saved to a database, making it persistent across application restarts.

[13] an open source development framework for writing internet applications in Java. see http://seamframework.org/ for more information

[14] http://www.eclipse.org/

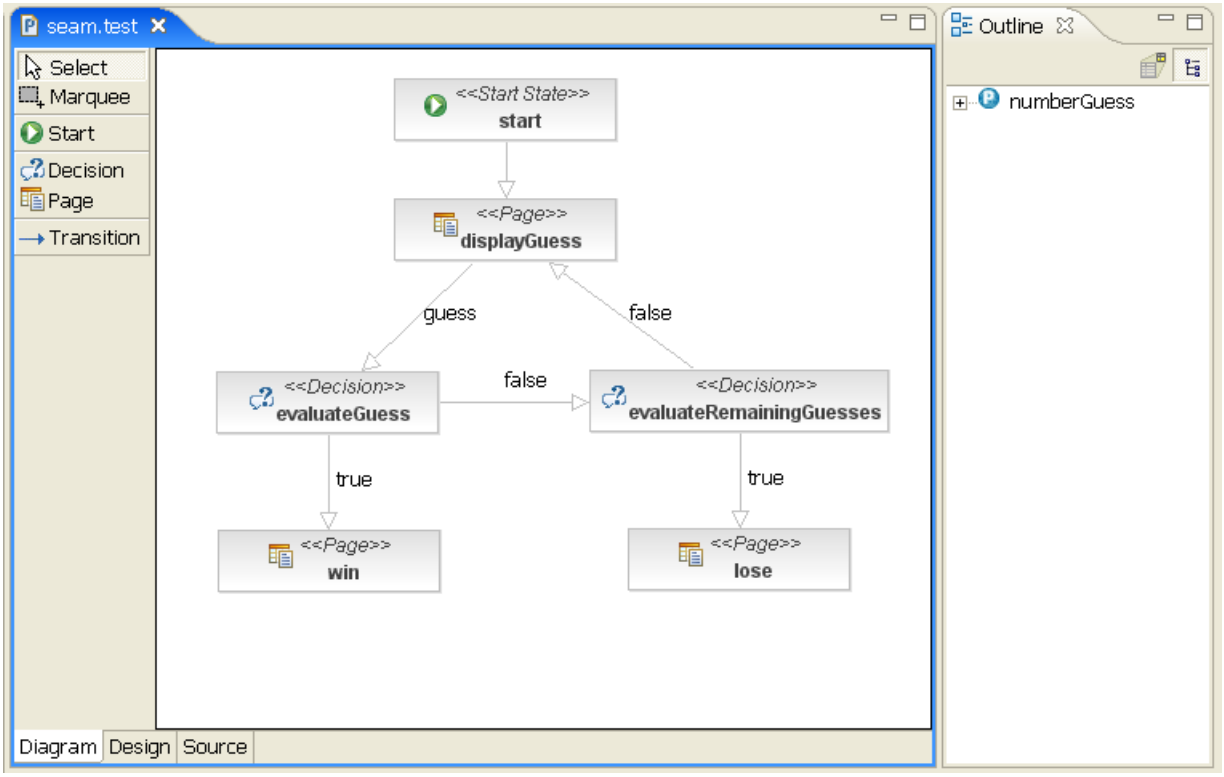


Figure 2.18: jBPM pageflow

Figure 2.18 shows a simple jBPM pageflow graph for a number guessing game. When the user starts the interaction, the system begins the pageflow in a start state, which displays a page to the user that lets him enter his guess. In addition to start state and pages, a pageflow uses state to make decisions, indicated by special nodes in the graph. In the example, a decision node checks whether the user guessed the correct number. If he did, he will be presented a page informing him that he won. If not, another decision node checks if he has guesses remaining. If yes, he is directed again to the previous page, if not, he will see a page telling him he lost the game.

Graphs for processes look very similar, only instead of page nodes they contain task nodes. Each task node can consist of a pageflow, and has an assigned actor, a specific user or system that may perform this task. Additionally, processes may include forks and joins, and, due to their persistence, they must have an explicit end state.

The support for processes involving multiple users, especially processes like games, where users interact with each other, is the greatest advantage of jBPM. Such processes require synchronization, which is very difficult to manually program over the stateless HTTP protocol. jBPM, however, relieves the programmer of this work, only requiring him to specify states, the decisions and transitions between them, and tasks, including pageflows and actors, while the system takes care of the creation of the Java code for executing the process, including synchronization.

Since jBPM generates Java code and does not directly execute the program, the programmer is also able to further edit the generated code. Roundtripping, the generation of a graph from Java code, however, is not possible.

2.3.5 ComiKit

ComiKit [5] is a good example for how greatly visual programming languages can differ in their visual formalisms, since it's entirely different from the languages in the previous examples. It is an imperative language based on visually represented state changes.

ComiKit is a constraint oriented system, a *simulation-* or *game-maker*, with an educational purpose mainly aimed at younger children, similar to the KidSim that was mentioned briefly in the introduction to chapter 2.3. However, the authors identified several problems with the graphical rewrite rules of previous works, and tried to overcome them with a language design based on comic strips.

The problems of earlier systems employing graphical rewrite rules are threefold.

The first problem is limited graphical programming expressiveness. Before-after rewrite rules are inflexible, in that both parts must have the same size and show the same place of the world[15] and the same objects. This tight coupling between the before- and the after-picture of a rewrite rule makes the expression of general rules impossible or very difficult, for example the rule "when the sun shines, a character smiles" can hardly be expressed, since the sun and the character may be anywhere within the world, and the character might be any character, while the rewrite rules require the depiction of a specific character.

The second problem lies in the grid-based world model of most systems, which requires grid-based rewrite rules, that have been shown to be very counter-intuitive when expressing left or down movements.

Finally, the third problem is also based on the grid-based world model. The model is responsible for very rigid world appearance and dynamics, coming from the requirements of objects to have fixed sizes, always be placed at discrete locations, and an inability to overlap. Movement is jagged, with objects only able to move from one grid cell to another.

ComiKit tries to overcome these limitations of traditional graphical rewrite rules by employing insights gained from the semiotic analysis of comics, since comics are a familiar medium, especially to ComiKit's young target audience. They are, like a program, a static representation of something dynamic (still pictures of an ongoing story), using sequences of panels as the basic temporal device, and have a rich vocabulary of signs to express some dynamic or abstract features, like lines to represent speed, voice balloons to represent speech or pain stars and hearts to represent emotions.

ComiKit employs event-based programming with *conditional comic strips*, where the first panel is the precondition for the actions in the subsequent panels.

[15] in constraint oriented systems, the game or simulation usually runs within a fixed space, referred to as the *world* or *game world*

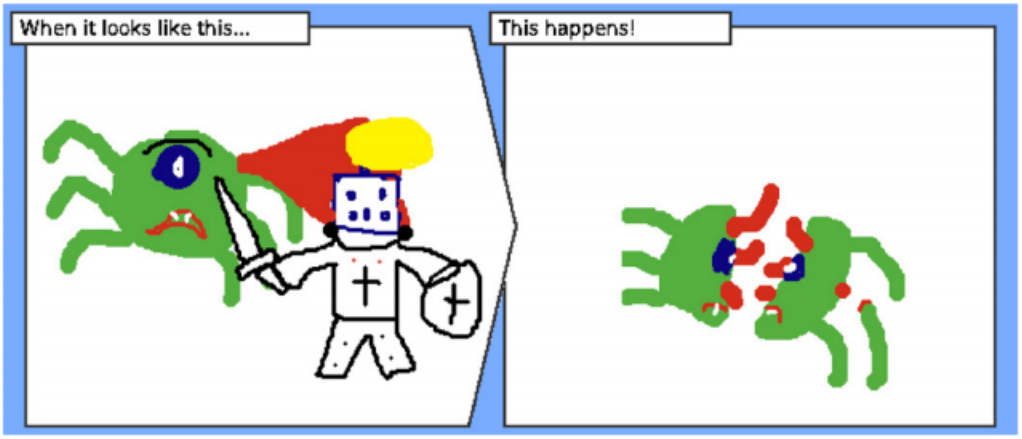


Figure 2.19: A comic strip event

Figure 2.19 shows a very simple conditional comic strip. In the first panel, the precondition, we have a knight and an evil monster. The overlap of both pictures symbolizes that they touch. In the second panel, the action following the precondition, we see the dead monster. When the program runs, whenever a knight touches a monster, the monster is slain.

The available preconditions are object type (e.g. a knight), picture matching (ComiKit uses an object-oriented system, where multiple pictures can be assigned as instances of an object), touching (overlap), key press (symbolized by a picture of a key), and (random) time interval, represented by a text box with the time at which the event should check its precondition.

Conjunction of preconditions can simply be expressed through multiple preconditions in the same panel. For disjunction, it is required to create multiple strips.

The available actions, following a precondition, are motion, picture change, speaking, and object creation or deletion.

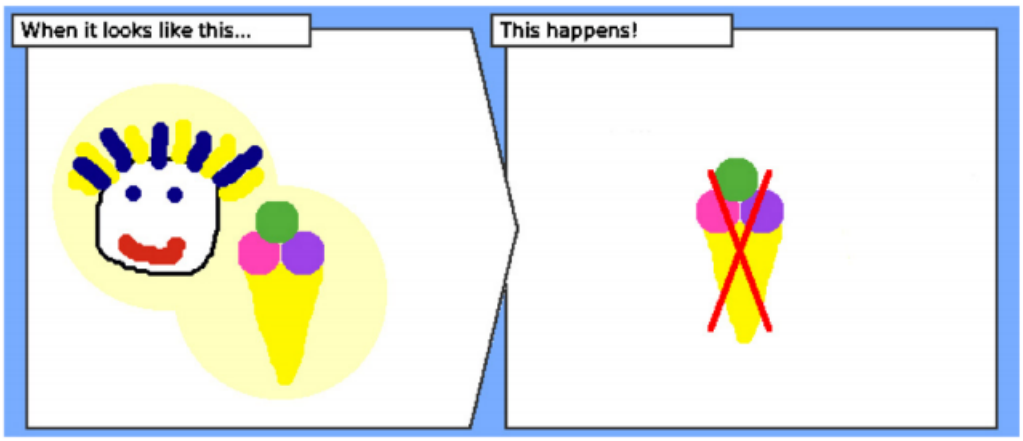


Figure 2.20: Another comic strip event

Figure 2.19 already has shown the absence of a grid in ComiKit's world model by allowing objects to overlap. Figure 2.20 shows how, despite looking similar to graphical rewrite rules, conditional comic strips are different. In the image we have a character and a cone of ice cream in the precondition, again with overlap symbolizing touch, and a cross through the ice cream cone, symbolizing object deletion, as the action. The event is simple - when

a character touches an ice cream cone, he eats it, and the ice cream is gone. But what the example shows is that the events are not as restrictive as graphical rewrite rules: different objects can appear in different panels, and the panels can show different parts of the world. Comic strip events are therefore more like general conditional expressions, with condition and body expressed using pictures of domain objects and contextual signs.

The earlier mentioned rule "when the sun shines, a character smiles" can easily be programmed in ComiKit with a strip that contains a shining sun in the first panel, and a smiling character in the second. Furthermore, if the rule should also have the precondition that only a sad character should change to a smiling character when the sun shines, the independence of the positions of the sun and the character can still be expressed, simply by placing either the shining sun or the sad character within an inner panel within the precondition panel.

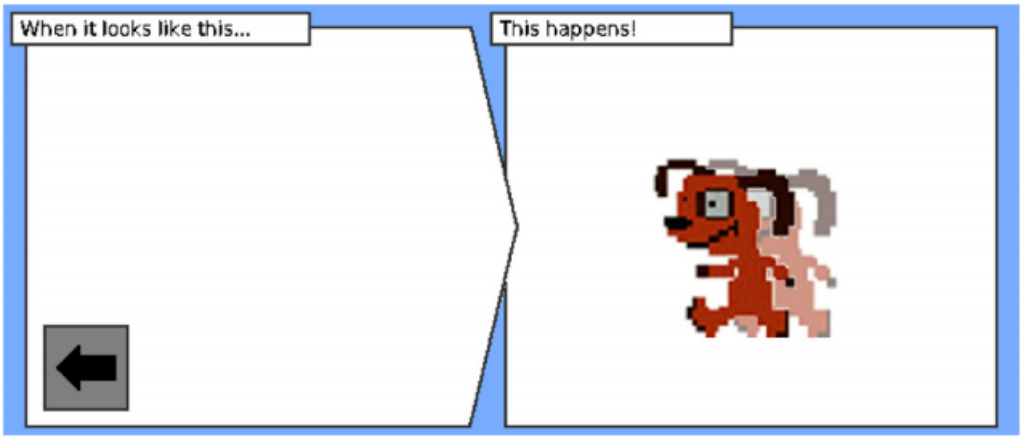


Figure 2.21: An interactive comic strip event

Figure 2.21 shows a strip for an interactive event. In the precondition panel a picture of the left-key symbolizes the respective key-press. In the action panel, a character moves to the left in response to it. Remember the problem of counter-intuitive representation of left-movement in grid-based graphical rewrite rules. This is overcome in ComiKit with a ghost image, showing the character's previous position below its new position, which was easily understood by the children in field studies with ComiKit.

The expressive power of ComiKit is relatively low, with important features of other programming languages, like variables, concurrency, or negation, missing. However, for its intended purpose of allowing children to create simple games or interactive storybooks, it is well suited, and it exemplifies the breadth of different visual techniques, and different application domains, that visual programming languages have ventured into.

2.3.6 3D-PP

The visual languages introduced so far all made use of two dimensions, which is quite natural, since the user looks at them on a two-dimensional screen. However, since enough processor power and graphics card support for three-dimensional graphics have been available since the early 1990s, some attempts to create three-dimensional visual languages have been made.

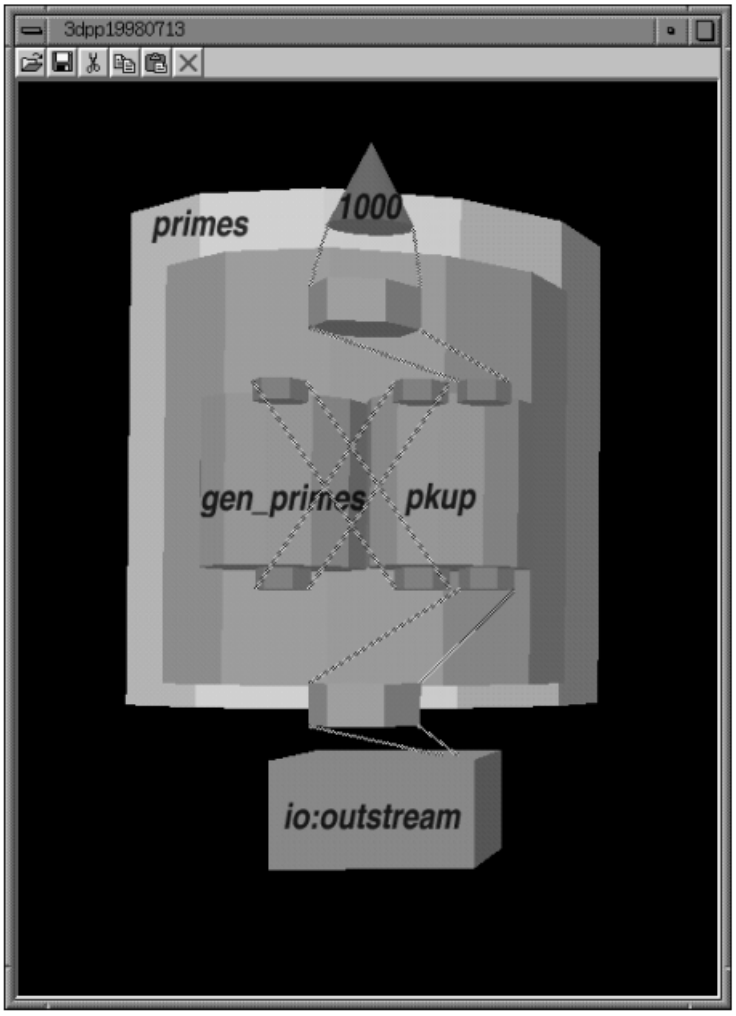


Figure 2.22: A 3D-PP program

One of the first programming languages using three dimensions was Cube [29], but it suffered from being more a three-dimensional rendering and execution environment for a textual language, lacking any graphical editing features.

A language that overcomes this and provides fully three-dimensional interaction, program editing and execution, is 3D-PP [30].

As their motivation for venturing into the third dimension, the authors of 3D-PP name the restriction of two-dimensional visual languages to be unable to handle more than toy programs in a single window, and that large-scale visual programs require too many pictorial elements. They also say that, since our world is three-dimensional, programmers will have an easier time working with a three-dimensional language, which makes it easier to grasp relative positions between elements and has a more effective three-dimensional animated program execution representation.

3D-PP itself is based upon a high-level declarative, concurrent logic programming language.

A program is built by combining and nesting three-dimensional programming elements corresponding to elements of the textual language, like atom, list, I/O data and goal.

The problem of using a mouse, which is a two-dimensional input device, to manipulate

the three-dimensional program, is solved with an extended three-dimensional drag-and-drop technique, employing a *restrictive plane*, that limits object movement to two dimensions parallel to the current view screen, and semi-transparency is used, so that dragged objects are never hidden from view.

3D-PP also uses information hiding: objects can be double-clicked, to move into them and display their sub-elements, or to hide them.

Figure 2.22 shows a three-dimensional program to calculate the 1000th prime number.

The outermost elements are the goal "primes", the atom "1000" and an I/O output stream, showing us that the program generates the 1000th prime number and writes it to the output.

Within the prime goal, we can see the elements that make up its inner workings: "gen_primes" is a goal that generates prime numbers, "pkup" takes the 1000 as input from the surrounding goal, then picks up the corresponding 1000th prime number from "gen_primes" and transfers that one back to "primes", after which it is taken to the outstream. The sub-goals of "gen_primes" and "pkup" are hidden here, but could be accessed by double-clicking the respective elements.

3D-PP also displays the execution of a program as a three-dimensional animation, by displaying the resulting objects when goals are reduced, and atoms, with their value, moving around between the elements whose input or output they are.

2.4 Visual Query Languages

The third and last domain, in which a huge number of visual languages can be found, is that of database querying.

To support the user not only in constructing queries, but to also help him visualize the structure of the data he is working with, most visual query languages do not only support visual query construction, but also come with some way of browsing the database contents and visually exploring the database schema, and a way to visualize the query result, which is why literature usually speaks of visual query systems (or short, VQSs), instead of visual query languages, to highlight their additional capabilities, which can be independant from the visual language the VQS is employing.

The application domain of VQSs is more limited than that of visual programming languages, which makes them more similar to each other and therefore easier to compare and classify.

In [9], some criteria for classification are given, which shall be introduced here.

Data and query representation

The first criterion is how the database and the queries are visually represented.

There exists a great multitude of different visualizations, but they can be grouped into four classes.

Form-based systems employ a visualization technique inspired by paper-forms or spreadsheets. The basic elements are usually elementary cells, with no nesting allowed.

The user can then fill out certain cell values. A form can represent relationships either among cells, among subsets, or the overall set. This representation is usually only little graphical, consisting mainly of text, ordered on a two-dimensional form. visKWQL can be considered form-based, although it also employs features from diagram-based systems like element nesting and uses colors to discern between element types.

Diagram-based systems are more graphical, and are based on graphs or diagrams. In diagram-based systems, information can be expressed through shape, position, magnitude, color, and other properties of graphical objects, and through connections or other relationships (like nesting) between them.

Icon-based systems use icons (stylized images or symbols) to represent both real and abstract concepts or actions. Queries are created by combining icons according to some spatial syntax.

Hybrid systems exploit the advantages of the other approaches by combining two or more of them. They either allow the user to choose between the different formalisms, or combine them into one representation, for example to display the database schema with a diagram and use forms to show details of an entry.

Result visualization

The next property of a VQS is the way it displays query results.

Here usually the same formalisms as in query and data visualization, namely form-based, diagram-based, icon-based, or hybrid representations, are used. Some systems however don't visualize the results at all and just return a textual output.

Interaction strategies

Another criterion, to compare VQSs by, is the way they handle user interaction.

User interaction can be divided into two activities: understanding the reality of interest, and query formulation.

Understanding the reality of interest refers to how the user can gain knowledge of the database he is working with, which means understanding the database structure or schema, and exploring the database content. VQSs employ several different strategies to help the user with this task.

Top-down strategies let the user first perceive general aspects, and then gradually more details. This is done by either a sequence of iterative refinements, or by some sort of zooming capability.

Browsing can be split into three different types. *Intensional browsing* is performed on the database schema, and can be thought of as meta-querying it, for example asking for all concepts with a given property. *Extensional browsing* lets the user explore the database content. The system can, for example, show him a graph of the entries in the database, or let him select an object in the database schema and show him the object's instances. *Mixed browsing* combines the former two methods.

Schema simplification lets the user build a view of the database resulting from aggregations and transformations of the database schema.

Query formulation refers to how the system lets the user create a visual query. The formulation can be supported through

Schema navigation: The user follows some path through the database schema, and selects the concepts involved in his query along the way. VQSs can further be distinguished whether they allow the navigation along arbitrary connected paths, hierarchically connected paths, or unconnected paths

Subqueries: The user formulates queries by composing partial results, either through composition of concepts or use of stored queries, which are either templates stored in the system, or queries saved earlier in the query session.

Matching: Queries are created by presenting the system with the structure of possible answers, which the system then matches against the available data. The structure can be presented either *by example*, where the user supplies an answer and the system attempts to deduce the structure from it, or through pattern matching, where the system provides some syntax to formulate regular expressions, which are then used to match the data against.

Range selection lets the user supply a range of values for certain attributes, most often with graphical sliders, with the system constructing a query, that then searches for data items with attribute values in the specified ranges.

These were the classification criteria concerning the visual part of VQSs, and the query representation criterion will be used to group the examples later in this chapter.

There are, however, some more ways that visual query systems can be distinguished by, which do not concern their graphical features:

Queryable data: One can compare VQSs according to the database sources they work on: relational databases, object databases, object-relational ones, unstructured data like XML, RDF or OWL ontologies, etc.

Textual language: Many visual query languages are based on some textual query language, which allows to distinguish VQSs by whether they are based on a language like SQL, XQuery, a prototypical language the authors have created specifically for their needs, or no textual language at all, if the visual language has complete visual syntax and semantics and a supporting database management system.

Expressiveness: Languages can be distinguished by whether they support all features of the textual language they are based on, or only some of them, or by what operations they support, for example what kind of joins. Furthermore, some VQLs only allow querying the database, others may also allow data transformation or construction.

Ease of use: How easy a VQS is to use is sometimes taken as a criterion to distinguish systems. However, its use is pretty unclear, since ease of use is a highly subjective matter, depending on such factors as the knowledge, skills, or personal preferences of a specific user.

One aspect that sets visual query systems apart from visual modeling or visual programming languages, is the amount of different types of users these systems have.

While modeling languages (in the field of computer science) and programming languages (excluding those with an educational purpose) are usually used either by experts or at least by knowledgeable programmers, the people using VQLs range from the expert database administrator, to the secretary that always uses one and the same query to create an employee list, to the hobbyist web programmer who wants to include a news feed on his page that reads updates from some XML data file on a remote server.

To develop better systems, it is therefore profitable to have some understanding of the possible users and their needs, and some research in that direction has been done [9].

One classification that can be made, is to group users into professional users, and the so-called *casual* users, who only occasionally interact with a computer, have little or no training on its usage, have no knowledge of textual query languages and can usually not be bothered to learn them, and are unfamiliar with the internal organization of a database.

This, however, is a very broad characterization, and a more precise classification can be useful.

The authors of [9] suggest the following:

Initially, users can be compared according to the type of computer training they have had.

Professional users have a job centered around operating computer applications and possess a variety of skills, from programming languages to database management systems.

Non-professional users however have not had any formal training on programming or query languages.

The non-professional users, who are bound to profit the most from easy to use, visual query systems, can then be classified further, according to:

Frequency of interaction with the system: *Occasional users* only seldom need to query the database. For them, iconic VQSs are the most suitable, because those don't require the learning of a specific data model. *Frequent users* access the database often, and therefore profit most from form-based systems, which often provide a set of predefined queries.

Variance of the queries: *Repetitive users* always use very similar queries, which makes form-based systems, where access is possible through simple menus, advantageous. *Extemporary users* however need to make queries that are always different from each other, so that they profit more from diagrammatic or iconic systems that provide easy navigation in the query space.

Structural complexity of the query: *Naive users* don't require many different operations, and therefore don't need the full power of a query language, making iconic languages, which provide immediate mechanisms for simple queries, ideal for them. *Sophisticated users* make queries using the full expressive power of a language, making diagrammatic languages most fitting for their needs.

Familiarity with the database semantic domain: *Familiar users* are acquainted with the data they are querying. *Unfamiliar users* do not know the details of a database's contents, and therefore profit most from iconic or diagrammatic systems, which can well express the internal data structure.

The first two points, frequency of interaction and variance of the queries, allow a user classification that refers to typical roles in organizations.

Frequent-repetitive users are clerks and secretaries, who usually interact with the systems through menus and forms, and rather choose a query from a predefined set than create a new one.

Frequent-extemporary users are managerial types, who often strive to become autonomous in the use of the system and therefore profit most from hybrid systems that allow them to choose an interaction strategy according to their skills.

Occasional-extemporary users, using the database very seldom but requiring a wide range of different queries, usually delegate the query formulation to more expert users, and would need a VQS that is very powerful and very easy, to capture their interest.

Finally, since hybrid systems combine the aspects of the other VQSs, they seem best for most users, since they allow the users to choose between different interaction modalities.

The next sections show some interesting examples of different VQSs. Chapter 2.4.1 introduces a number of form-based visual query languages, chapter 2.4.2 diagram-based ones, and chapter 2.4.3 shows a small number of hybrid systems.

2.4.1 Form-based Visual Query Languages

2.4.1.1 QBE

QBE [3], short for Query-By-Example, was the very first visual query language.

It was developed in the mid 1970s, as an alternative to the textual structured query language SQL. Its design goals were to provide a convenient, unified, high-level language to query, update, refine and control a relational database.

The language should require only little prior knowledge from the user, and consist of only a small number of concepts, making it easier to learn than a textual language.

Furthermore, the language should capture the relational database model's symmetry, simplicity and neutrality, by mimicking manual table manipulation.

The chosen approach, to fulfill the design goals, is to provide the user with two-dimensional skeleton tables, in which he can then fill in an example of the solution he searches. He can provide *constant elements*, which are values, that the solution's attributes are required to have, and *example elements*, which act as variables. He can also use a number of special instructions, for instance for sorting the results, printing them out, or retrieving the count of the resulting entries.

TYPE	ITEM	COLOR	SIZE
	P. ROD	GREEN	

Figure 2.23: A simple QBE query

Figure 2.23 shows a simple QBE query. When the user starts the query formulation, only the empty outline of a table is displayed. In the field in the upper left corner, the user can

enter a table name, in this example “Type”. The system will then fill in the table’s column headings with that table’s attribute names, or let the user enter them himself.

After the table skeleton is complete, the user can proceed to fill in example values. The “P.” in the “item” column stands for “print”, and indicates the desired output. The underlined “rod” indicates an example value, that is an example for a possible output, used like a variable. The “green” finally is a constant value, a restriction or required condition, on the output. The query therefore will print out all items that have a green color.

Note that for brevity, if a variable is not used anywhere else in the query, it is usually omitted and only the “P.” remains.

As stated before, QBE aims to provide a unified style to querying, updating, etc., which is why a query’s output is presented in the same two-dimensional format as the constructed query. In the above example, if the query is executed, the result would be a new table with just the item column with the names of all green items in it. However, a result can also be a whole table, if the user adds the “print” instruction to multiple cells.

Qualified retrieval in QBE can be accomplished by adding inequality operators to constant elements (for example, “>100” in the “Size” field of Figure 2.23, to find all items that are larger than 100cm) or by partially underlining elements, which expresses partial string matching (for example, “*P.*$\underline{X}$ visual $\underline{Y}$” would return all items that contain the word “visual”). Negation can be expressed through a special marking of cells or elements.

EMP	NAME	SAL
	P.	$\underline{S1}$
	JONES	$\underline{S2}$
	NELSON	$\underline{S3}$

CONDITIONS
$\underline{S1} > (\underline{S2} + \underline{S3})$

Figure 2.24: QBE query with a condition box

Arithmetic expressions can be used on all attributes with numerical types. To express more complicated expressions, QBE provides a *condition box*, which allows the user to enter various conditional expressions, involving the variables defined in the query.

Figure 2.24 shows the use of such a condition box. The query will return the names of all employees whose salary is greater than the combined salaries of Mr. Jones and Mr. Nelson.

In addition to instance data, QBE can also query the database schema. For example, a user could enter the print instruction in the column below the table name, and the system would return the names of all tables in the database. A print instruction in the same column’s heading, behind the table name, would return all the column names of that table.

Insertion, deletion, and update of database entries is possible in the same simple way, by replacing the print instruction “P.” with “I.”, “D.”, or “U.” instructions. The insert and delete instructions can also be used to create, expand, or drop entire tables.

QBE was a seminal work that inspired many visual languages. But, apart from a use in few software packages like Microsoft’s Access, it has mostly vanished from today’s world.

This is due to QBE not having the same expressive power as its textual counterpart SQL. It is not relationally complete and has only restricted support for aggregations and groupings [31].

This has led to research going into other directions and an abandonment of QBE. However, about 20 years after QBE's initial conception, there have been some attempts to solve its problems:

SubQuery-By-Example [31] is an orthogonal extension of QBE that reaches relational completeness and fully expressive aggregations and groupings by including parametrized sub-expressions and new operators for set and aggregation expressions into the QBE core.

QBEN (Query By Example For Nested Tables) [32] takes the QBE philosophy and applies it to an approach using nested tables to represent data. The authors describe it not as a generalization of QBE, but as a "variation that takes advantage of the capabilities of modern software technologies".

QSByE (Querying Semistructured data By Example) [33] finally is a QBE implementation based on nested tables, that extends the relational algebra with features that support nested and irregular structures, to allow the querying of semistructured data instead of relational databases.

2.4.1.2 EquiX

EquiX [34, 35] is a search and query language, combining pattern matching, quantification and logical expressions, to query XML data and meta-data.

It is aimed at users who need to query XML data, but have no knowledge of any (textual) query languages.

Providing an easy to use, intuitive query language for those users was the primary design goal of EquiX, which is why some expressive power was consciously forsaken to reach higher simplicity of the language. The authors see their language mainly used to search for information, wherefore they argue, that a search query need not restructure documents, but merely return them, and the language can retain more simplicity with the omission of data construction capabilities.

EquiX' principle is to apply the Query-By-Example philosophy to XML data, by letting the user construct query forms by interactively exploring given DTD files.

Query forms in EquiX can be seen as templates, in which the user fills in an example of his desired query result.

After he chooses an XML catalog to query, the system displays the root node in the query form. Elements can be explored by clicking on them, which opens a view of their attributes and children elements.

The user can then iteratively select attributes and children elements, fill in text fields to place constraints (similar to QBE), specify quantification, and choose which elements should appear in the output. The system then takes this example document, matches it against the documents in the selected catalog, and returns all documents that fit, automatically generating a new DTD from the query form.

The text fields the user fills in may contain constants, variable definitions, inequality conditions, and the aggregate functions count, sum, average, maximum, and minimum.

All other elements may be quantified in EquiX, through the conditions *one is*, *none is*, *all are*, and *not all are*.

The query forms themselves are displayed in a way that should be familiar and easy to work with for most users, using a tree-view, similar to an operating system's file browser.

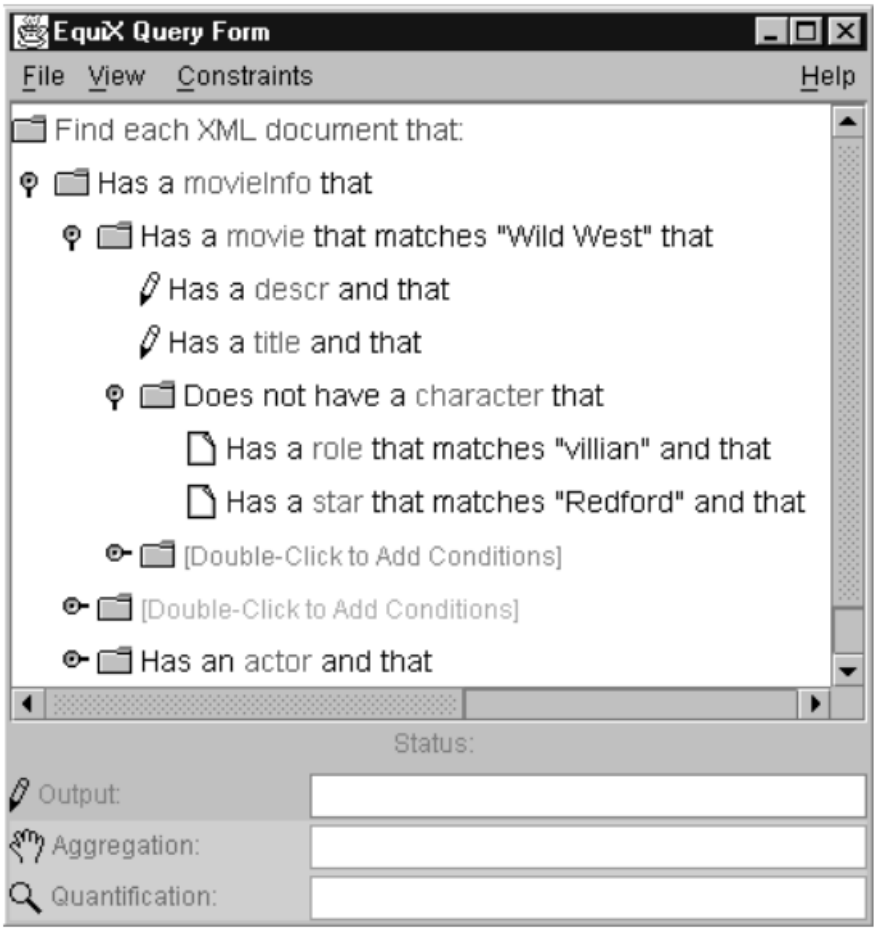


Figure 2.25: An EquiX query form

Figure 2.25 shows a query form. XML elements are indicated by folder symbols, attributes by file symbols, with those selected for appearance in the output having a pen symbol instead. At the bottom of the window are boxes where aggregation or quantification expressions can be entered for a selected attribute or element. The query would return the descriptions and titles of all movies in which Redford is not a villain.

EquiX' simplicity can be easily seen in the example: the query can just be read from the form without any ambiguity. Due to this fact, the authors of EquiX argue, that form-based query languages are better suited for naive users than diagram-based ones, since their graphs are unintuitive for non-scientific users.

2.4.1.3 BBQ

BBQ (Blended Browsing and Querying) [36] is a XML query language that takes the same approach as EquiX, visualizing XML data with trees similar to file browsers. However, in BBQ, expressive power of the language is more important than simplicity, so that it has a number of features that are missing from EquiX.

As is already apparent from the name, BBQ places high importance on browsing, or visualizing the data sources the user wants to query.

To reach this goal, BBQ allows the user to open arbitrary XML files, and displays the files, and the accompanying schemata (DTD files), both as trees in a multiple-window interface. It also allows working with XML files that have a loose or incomplete schema, by applying a DTD inference mechanism to create a complete schema.

The user can then explore, or *browse*, both the data and the schema by drilling down the tree (like in EquiX, XML elements can be clicked in the tree, to "open" them and view their attributes and children) and placing conditions on elements in the data or schema tree, to restrict the number of shown elements.

Since BBQ supports the visualization of DTDs, it needs to represent its structural operators, like choice and sequence. These are implemented straightforwardly as special tree nodes, with the operator's arguments contained as children within the tree node representing the operator.

Next to the comprehensive browsing capabilities, BBQ supports result construction. It does not just allow the user to select which elements should appear in the output, but he may create a completely new XML structure. Note however, that this only allows the creation of results with a new schema, for subsequent browsing. BBQ does not support the restructuring of the original documents like a XML transformation language would.

In BBQ, the user takes part in *query sessions*. Each session consists of four parts:

1. The user sets constraints on the data sources. BBQ allows the incorporation of data from multiple XML files in one query.

2. A tree representing the query result schema is created. This process is supported through a drag & drop functionality. The user can simply drag XML elements or attributes from the schema tree to a special window and construct a new tree there. He can also manually add new elements.

3. The system translates the query into a textual query language[16] and executes the query.

4. The system creates a DTD for the result, and displays both the query result schema and data.

[16] called XMAS

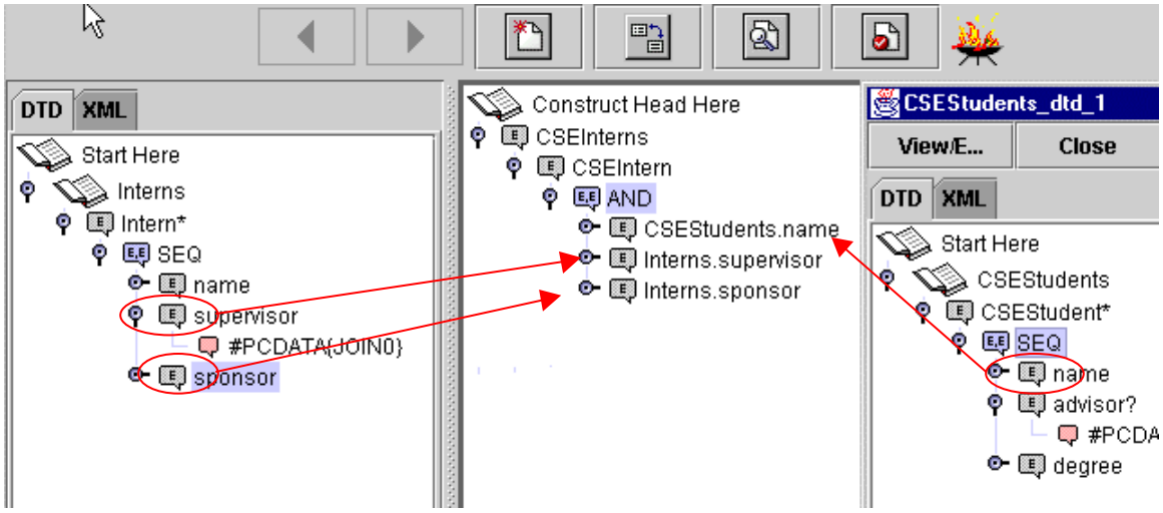


Figure 2.26: Constructing a query in BBQ

Figure 2.26 shows result schema construction. The middle tree is empty at the beginning of a session. To the left and right are two source schema trees, from which the user dragged elements to the result tree, indicated by the red arrows.

BBQ supports some more features that especially make working with large data sources easier:

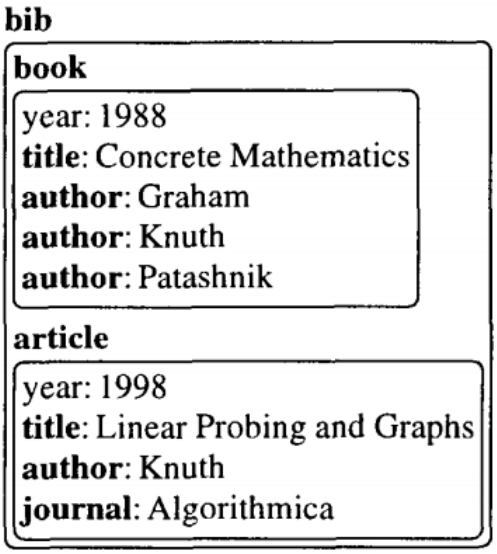
- It is possible to query a XML element only by its name, without specifying its exact position in the hierarchy. This is done via the *vertical wildcard operator*, represented as a tree node labeled "any". The user can insert an "any" node into a DTD tree and drag & drop another node on it, and BBQ will return all occurrences of nodes with that name, when the same node is subsequently dragged to the result tree.
- Query refinement is also supported. Users can use the result from one query immediately as source data for a new query. When incorporating multiple sources in a query, it doesn't matter whether those sources are actual XML files, or previously obtained query results.
- Finally, BBQ supports a history function, which lets the user return to a previous cycle in the query session.

2.4.1.4 Xing

Xing (XML in graphics) [37, 38] is a language that takes a radically different approach to form-based XML querying and transformation from the previous examples, and has visually much more in common with VXT (see chapter 2.3.2) and visKWQL, since it also employs nesting of elements.

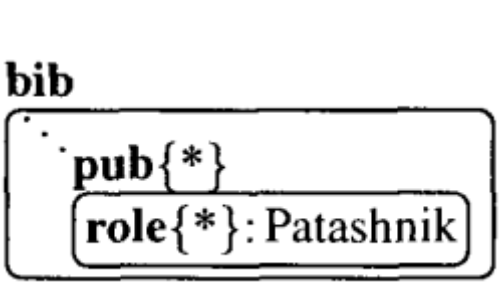
Xing's goal is to let users draw examples of the documents they are interested in. The system should therefore be as simple and intuitive as possible, avoid XML syntax, employ pattern matching and facilitate restructuring.

XML elements are represented in Xing as nested boxes, with the tag printed above the box, and the element's children nested within the box, together with their attributes (see Figure 2.27a for an example).

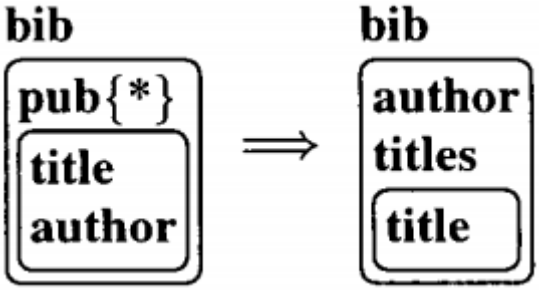


(a) XML visualization

(b) A simple query



(c) A deep query



(d) A rule

Figure 2.27: Xing examples

Queries in Xing are based on *document patterns* that describe properties of the requested information and, optionally, how it should be restructured.

Patterns consist of constants, which must appear in the search result to match the pattern, and variables which can be used to bind data for further processing or constraints.

Figure 2.27b shows a very simplistic query in Xing to demonstrate its ease of use. The query will simply return all book elements within a bib element. Note that the user is supported in the query construction when a DTD for the XML he is querying is present. In this case, the system will present the user with a list of all possible children elements and attribute names to choose from, when he edits a query element.

Xing uses regular expressions to let the user further specify elements and attribute values. Stars can be used to express "any" element (for example, if "book" would be replaced by "*" in example 2.27b, then not only all books, but all direct children of the bib element would be returned), and a dashed line of three dots represents an arbitrary nesting (queries containing it are called *deep queries*). Variables can be introduced for all elements.

Figure 2.27c shows a more complicated query. It finds all children elements of bib, at any depth, and assigns them to the variable "pub". It also finds all elements that have "Patashnik" as a value for any attribute, and assigns that element to the variable "role".

Finally, Xing allows restructuring of documents with rules. Rules consists of a query part and a construct part. The construct part may restructure the elements (specified directly or through variables) of the query part, and introduce new elements.

See Figure 2.27d for an example of a rule that searches all publications in a bibliography and creates a list of authors and their titles. For each occurrence of an author, a new element is automatically created. For each such author element, a new element "titles" is then created, containing all the titles that author has contributed to.

Furthermore, Xing allows the expression of universal and existential list patterns, a count-aggregation, conjunctions, and cross-referencing, giving it a high expressivity, all the while maintaining a very simple and easy to understand visualization scheme.

2.4.2 Diagram-based Visual Query Languages

2.4.2.1 G

G [39] was among the first diagram-based visual query languages, and probably the first that received widespread recognition, since, unlike textual query languages at the time, it had complete support for recursive queries on cyclic data.

The motivation behind G was the simple fact that most data consists of objects that stand in some relationship to each other, and, through these connections, form a graph structure.

A visual query language based on graphs would therefore be amenable, since it preserves this structure.

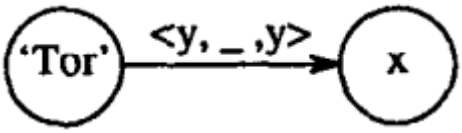
Furthermore, objects often have such relationships, that the resulting graph structure contains cycles. This cyclic structure would make queries like the transitive closure of the graph, or other queries involving recursion, interesting. However, textual query languages did not support the notion of recursion or transitive closure, which made the development of G worthwhile.

G expresses queries with labeled, directed multigraphs, that is graphs that can have multiple connections (in both directions) between each pair of nodes, and where each connection can have some label.

Labeled, directed multigraphs can be seen as relations, so that G can be considered a relational query language.

Through the labels however, G supports recursion, since those labels can be regular expressions over tuples of variables and constants. This allows to express, for example, a connection of arbitrary length in a query, so that one can formulate queries in G that would be impossible to express in relational algebra, even if it would be extended with a transitive closure operator.

Theoretically, the query formulation in G is done by drawing circular graph nodes, entering constant values or variable names into them, connecting them with directed edges, and adding labels that consist of regular expressions to the edges. In G's prototype however, queries are entered into the system through a string representation of the multigraph.



(a)

(b)

Figure 2.28: Queries in G

Figure 2.28 shows two simple queries in G, based on an airline database. The nodes represent cities with airports, the edges represent flights between the cities. The edge labels indicate the flight's airline.

In Figure 2.28a the starting city is Toronto and the end point may be any city, indicated by the variable x. The connection between the two cities is a regular expression of the flights that connect them. The first and third flight may be with any airline, however both flights must be with the same airline, indicated by the variable y. The second flight is with a completely arbitrary airline, indicated by the underscore. Thus, the query searches for all cities x and all airlines y, where x is reachable through exactly three flights from Toronto, where the first and last flight are with the same airline.

The second example, in figure 2.28b, shows the use of arbitrary length connections in a query, indicated by the dotted arrows, labeled with regular expressions that contain the "+" operator, which represents an arbitrary, non-zero, number of repeats. The query will find all cities x and all airlines y, where x is reachable from Toronto with a stop in New York, where an arbitrary number of Air Canada flights connects Toronto and New York, and an arbitrary number of flights with airline y connects New York and the destination city x.

2.4.2.2 GraphLog

GraphLog [40] is a query language for deductive databases, based on the textual logic language DataLog.

Its visual representation is strongly based on G, with data represented as directed, labeled multigraphs, where edge labels are regular expressions. It also introduces some additional features like negation.

GraphLog's greatest difference to G comes from the fact that it is not a relational query language, used to find certain elements in the database, but that it operates on a deductive database, because of which its intention is not to find elements, but to deduce new information. It does this by searching for patterns in the database, which must either be present or absent, and then creating new relations.

The user creates a query graph, similar to G, with the addition of a *distinguished edge*, and the restrictions, that no node must be isolated, and no explicit recursion must be present. The graph itself is the pattern that will be searched in the data graph when the query is

executed. The distinguished edge is a new relation, that will be inserted into the data graph when the pattern is found.

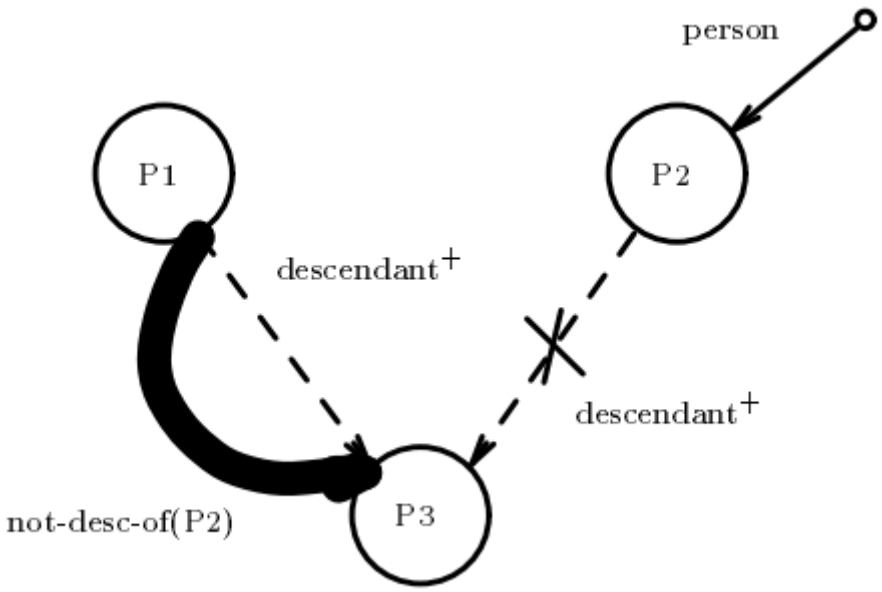


Figure 2.29: A GraphLog query

Figure 2.29 shows a GraphLog query that demonstrates the use of regular expressions, negation, and the distinguished edge.

It contains a pattern, consisting of three persons P1, P2 and P3. P3 is, indicated by the label, a descendant of P1, and, indicated by the crossed arrow, representing negation, not a descendant of P2. If such a pattern of persons is found in the database, a new relation "not-desc-of(P2)" will be created, visualized through the distinguished edge, marked as a thick arrow in the image. The query thus marks all descendants of P1 which are not descendants of P2.

Unlike the G prototype, GraphLog supports direct graphical query manipulation and the display of database graphs. The user can cut and paste graph elements, edit the text of nodes and labels, scroll over large graphs, and reposition and reshape nodes and edges.

2.4.2.3 Good

After relational and deductive databases, Good [41, 42, 43] applies querying based on graph-pattern matching to object databases, which, by their nature, lend themselves ideally for diagrammatic representation. It does not rely on any specific textual language, but has an underlying abstract machine, so that interfaces to different concrete database management systems are possible.

Good provides some interesting and powerful new features over the two previous examples (G and GraphLog).

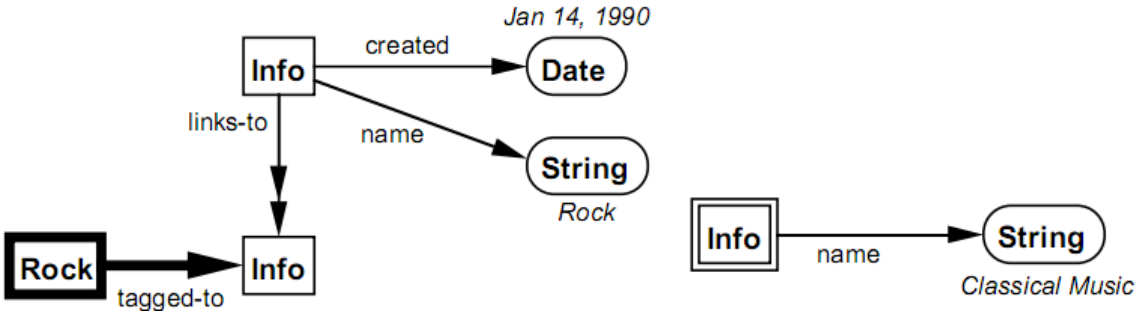
First, it does not only allow querying the database (like G), and displaying database instances (like GraphLog), but also supports schema visualization.

A database schema is represented as a labeled, directed graph, in which the nodes represent classes of objects, and the edges represent relationships between those classes, or properties that can exist between objects of those classes. The different types of classes are represented through different shapes: rectangles for abstract classes (which are usually user-defined), ovals for basic classes (which are system-defined, like String or Number). Database instances are visualized in the same way.

A second new feature of Good is user support during query construction, similar to the error prevention in visKWQL. Good queries are graphs that conform to the scheme graph of the database. The system enforces this, by allowing query construction only through identifying, copying and duplicating nodes in the scheme graph. This way, it is impossible for the user to construct an illegal query pattern.

Finally, Good can act as a complete database manipulation-, and not just query-, language, supporting database browsing (where nothing is altered), querying (where a transformed database graph is returned), restructuring (where the database schema is changed, but the content stays unchanged), updating (where the content is changed but the schema stays unaltered) and meta modeling (where Good accesses the scheme whose instances are valid schemes of the underlying data model).

It takes an approach similar to the edge creation feature of GraphLog, where a distinguished edge is added to the search pattern, which will be created when the pattern is found. In Good, a query pattern can contain additional elements to represent actions, which allow the creation and deletion of nodes and labels, abstraction[17] and method calls.



(a) Node addition (b) Node deletion

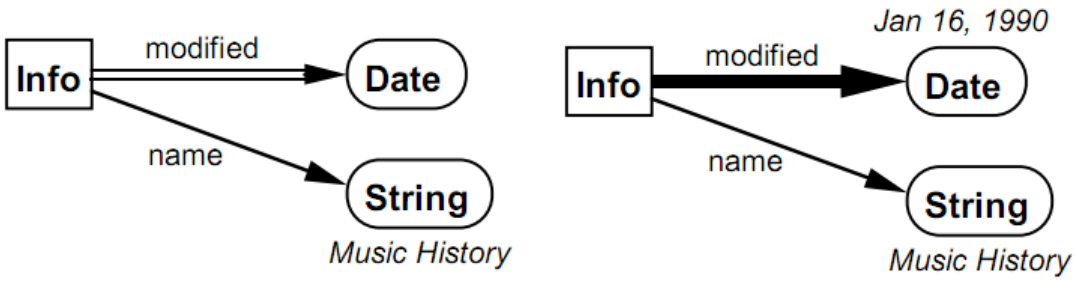
Figure 2.30: Node manipulation in Good

Figure 2.30a shows node addition in Good, represented by a node with a thick outline. The example searches all Info nodes in the database that are linked to an Info node created on January 14, 1990, with the name “Rock”. If such an Info node is found, a new node, “Rock”, will be added to it, and the connection between them will be labeled “tagged-to”[18].

Figure 2.30b shows node deletion. The node to be deleted has a double outline. The example will delete all info nodes whose name is “Classical Music”.

[17] defining new nodes in terms of properties of a base object

[18] The double-headed arrow represents a multi-valued connection, meaning that an Info node can be linked to an arbitrary number of other Info nodes



(a) edge deletion

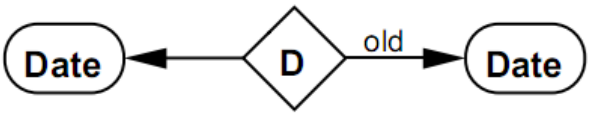
(b) edge creation

Figure 2.31: Edge manipulation in Good

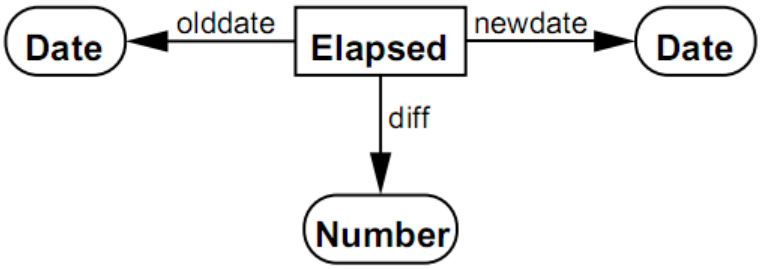
Figure 2.31 shows an update to the database. First, in Figure 2.31a, the edge between an Info node and its modification date is deleted (represented by an arrow with double outline). Then, in Figure 2.31b, a new edge (represented by the thick outline) is added to the date January 16, 1990. Note that a node or edge addition in Good can either introduce a new node or edge, or represent an update to an existing one.

A query pattern, together with the action defined in it, is called an operation, since it results not only in a query, but a graph transformation. A number of subsequent Good operations can thus be thought of as a program which gradually transforms the database.

Operations can also be grouped together into methods, which act similar to the methods of conventional programming languages. Methods have a specification, containing the method name, an associated node label, and parameters. They have an interface that specifies "side effects", which are nodes or labels that the method introduces to perform intermediate operations. Finally, methods have a body, which is a sequence of operations.



(a) Method specification



(b) Method interface

Figure 2.32: A Good method

Figure 2.32 shows parts of a Good method to compute the number of days between two dates.

First, the method specification is given (Figure 2.32a), in which the diamond labeled "D" represents the method name. D can be called on any two Date nodes.

In Figure 2.32b the method's interface is given. The interface shows the user what the method does (taking two dates and computing the difference between them) and allows him to use it, without requiring him to know how the method body (which is omitted here) looks. Note that nodes and edges (like "olddate", "newdate", etc.) that appear in the interface but not in the database instance, will not appear in the output.

Good also supports negation and recursion, which makes transitive closure computation possible. Furthermore, it has support for several object oriented aspects like complex objects, encapsulation and inheritance.

2.4.2.4 XML-GL

The object database query language G-Log [44] took many inspirations from Good, but combined it with first-order logic, and introduced the notion of rules. Rules in G-Log are similar to methods in Good, insofar as they can describe a graph (and therefore database) transformation. They can be combined to sets, in which case all rules are applied to the source data, or to sequences, in which case the rules are chained, meaning the output of each rule serves as source for the next.

The Language WG-Log [45] was formally based on G-Log, and applied its graph-based visualization schema and its notion of rules to query the Web. More exactly, it could query first generation web data, namely HTML documents and link structures [46].

Its successor, which shall be introduced here in some more detail, was XML-GL [47, 48], which applied the principles of G-Log and WG-Log to second generation web data, XML. It was the first visual language to address full XML.

Like most graphical query languages, XML-GL too uses the same graph-based formalism to both visualize the data and the query.

What was pretty new in XML-GL was that it is not, like its predecessors, based on some textual query language, but it is a complete language in its own right, with both syntax and semantics defined in graph structures and operations. This was a necessity, since during its conception, no textual XML query language with full expressive power was yet available. Today's preeminent textual XML query languages appeared as official W3C recommendations only in the same year (XSLT in 1999 [25]) and eight years later (XQuery in 2007 [49]).

The motivation behind XML-GL's design was that the hierarchical structure of a XML document corresponds to a tree, or to a graph when references between elements are included. This makes it possible to see querying as locating a sub-graph in a bigger graph, lending itself to easy query construction by copying and pasting nodes and arcs from a schema graph. Finally, document transformation can easily be rendered by constructing a new graph.

XML-GL had an extensive number of design goals, which the authors formulated not only for XML-GL itself, but as goals which every XML query language should aim to reach:

1. The language should be able to query both valid and well-formed XML documents

2. Queries to whole websites should be possible, including links between documents
3. The same query paradigm should be able to be used to query XML data, schema and meta-data
4. The language should support information extraction through pattern matching and element manipulation primitives
5. The language should support reshaping of documents, by both allowing to specify new elements within a document, and new links between documents
6. It should support regular expressions on paths, with the possibility of recursively following paths of arbitrary length
7. Arbitrary computation on numerical content should be possible
8. Both an unordered and an order-sensitive query interpretation should be available
9. The language should also be able to compute approximate results, which are only similar but not equal to results that fully satisfy the query
10. Queries should be readable, with the DTD of the result immediately apparent

XML-GL includes support for most of these features, and thus allows a wide range of visual queries, like selecting portions of input elements based on existential conditions and comparison predicates, expressing joins, creating arbitrary new documents from selections and new elements and relationships, applying arithmetic, aggregate and grouping functions both in the selection and the construction of elements, and computing set operations like union, difference or Cartesian product.

Its graphical data model consists of three building blocks, similar to those of its predecessors: *Objects*, represented as rectangles, indicate abstract items, or aggregations of *Properties*, represented as circles connected to the objects they refer to, which are representable values, like strings or numbers, and possess a name and a type, both represented as labels of the circle. Finally there are directed *Relationships*, represented as arcs between objects, which indicate semantic connections like containment or reference.

In addition to these, there are representations for special operations, like dotted edges representing negation, special nodes representing lists and indexes, and special labels for aggregation, arithmetic computation, predicates, ordering, mutual exclusion and the "Star arc", which represents all possible paths between two elements.

Each XML-GL query consists of two graphs, each of which can in turn consist of two parts, making a full query similar to SQL's *from-where-select-create view*.

1. The *extract* part (matching SQL's *from* part) allows the specification of target documents and document elements
2. The optional *match* part (corresponding to SQL's *where*) specifies logical conditions on the target elements, which they must satisfy
3. The *clip* part (similar to *select* in SQL) specifies the elements from the *extract* part, that satisfy the *match* part, that should actually appear in the query result

4. The optional *construct* part (*create view* in SQL) finally allows the creation of new elements or links and the restructuring of information local to a given element

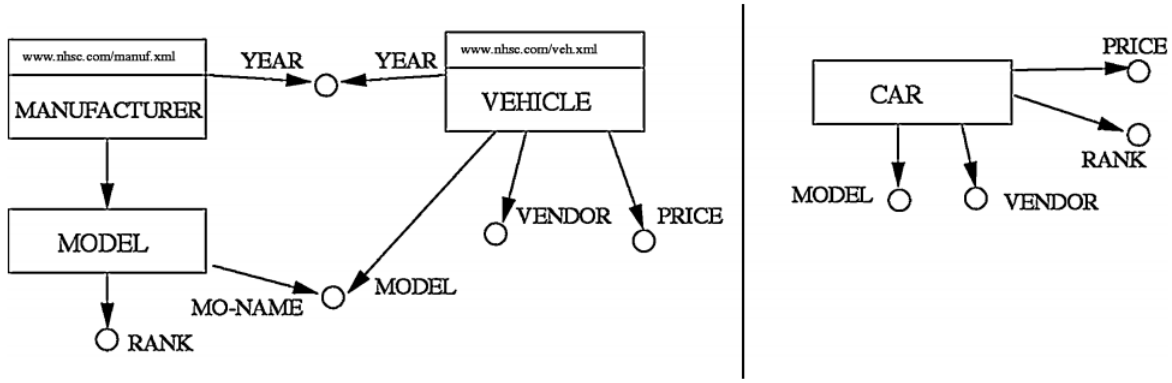


Figure 2.33: A XML-GL query

Figure 2.33 shows a query that includes all four parts.

The graph on the left side consists of the *extract* and *match* parts. It specifies two XML files, and the XML elements Manufacturer, Model (which is a child element of Manufacturer) and Vehicle, and a number of properties like Year and Price. The conditions of the *match* part in the example are the shared properties for year and model name.

The graph on the right hand side represents the *clip* and *construct* parts. The *clip* part preserves the properties Price, Rank, Model and Vendor from the *extract-match* part, while the graph's *construct* part introduces a new Car element, to which the properties get assigned.

The whole query thus searches for all vehicles which were built in the same year as some manufacturer has built a car, and share the same model name as some model the manufacturer has built. If such a pattern is found, a new Car element, with the appropriate properties, is created and returned as the query result.

Note that the aforementioned features of XML-GL, like aggregation functions or arithmetic computations, may be present in both graphs.

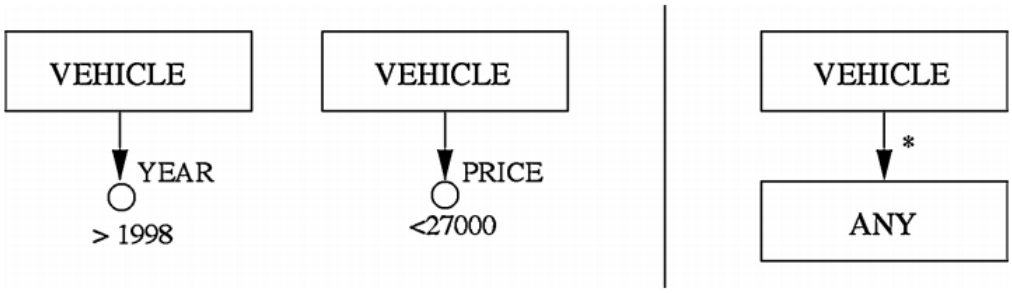


Figure 2.34: Another XML-GL query

Figure 2.34 shows a query involving numerical attribute predicates and a union (represented by two graphs with a shared element) on the left side, and the "Star arc" and special

"Any" node on the right side. The query first selects all vehicles that were built after 1998 and have a price lower than $27000. It then simply retains those vehicle nodes, including all their descendants (represented by the "Any" node), in the result. If the star would be absent, only the direct child elements of vehicle would be retained.

A few years after the conception of XML-GL, when the first drafts of the textual XML query language XQuery [49] had appeared, the visual XML query language XQBE was introduced [50]. Its graphical representation looks almost identical to that of XML-GL, but it introduced a few new capabilities, made possible through the underlying XQuery language, like joins between XML documents, conditions in the construct part, and renaming and projection of elements.

2.4.2.5 visXcerpt

visXCerpt [51, 52, 53, 54] is a visual language based on Xcerpt [55], a textual, rule based deductive language for querying and constructing XML data (an XML transformation language).

It is introduced in this chapter due to its query capabilities, however, like most XML transformation languages, it actually transcends the line between query and programming languages since it is Turing-complete.

One feature that visXcerpt shares with the KWQL Query Builder, introduced in part 3.3 of this thesis, is its realization through DHTML, making it not require any special editor or run-time environment, but making users able to edit and run visXcerpt program from within their web-browser.

What makes visXcerpt special, compared to other hybrid text and visual languages, is that it does not translate between the visual and textual representations of a program, but that visXcerpt is a visual rendering of the textual Xcerpt language.

This ability comes from several features of Xcerpt: first, and most important, is that Xcerpt programs themselves can be written in XML. This allows the application of CSS style sheets, with which one can specify how the web-browser should render HTML and XML documents, directly to Xcerpt.

The other features of Xcerpt, which support its visual rendering, are its pattern-based nature, rule based queries with clear separation of condition and result, and the fact that variables are specified by position and not by navigation path. Due to these, the visual rendering looks quite natural and is easier to grasp by novices.

visXcerpt conceptually consists of a static, a dynamic, and a server-side part.

The static rendering of the Xcerpt language with CSS style sheets was already mentioned[19].

Another part of DHTML, Java Script, was used for the dynamic part of visXcerpt, to enable the editing of the visual program.

To edit the program, the user can select building blocks within a template, and copy them to his program. This has the advantage, that the system can decide to allow or prohibit

[19] later versions of visXcerpt made use of an extension to ordinary CSS, called CSS^{NG} [54]. It is still an open issue whether new versions of CSS will offer all features needed for realizing visXcerpt and similar visual languages without making it necessary to rely on languages like DHTML

the insertion of the templates at the position the user has specified, to prohibit the user from creating syntactically incorrect programs, similar to the KQB's error prevention feature.

The dynamic part also includes the feature of *information hiding*, which has also been adopted by visKWQL, and allows the user to click on parts of the program to fold them together and hide their inner details, to make even large programs clear and allow them to fit on the screen.

The third part of visXcerpt's implementation is a server-side application, which interacts with the client-side editor, to allow the user to save and load programs, and of course to execute them through a Xcerpt implementation[20].

The actual rendering of visXcerpt programs is akin to the structure of the textual programs in their XML form: XML documents consist of nested elements, each with optional attributes. Similarly, the visual program consists of nested boxes, each holding textual values or special icons for the element's attributes, in addition to boxes for child-elements.

Xcerpt (and visXcerpt) were designed to query both the normal and the semantic web, which requires the support for graph structures (for example RDF data or hyperlink structures). These are supported through references in Xcerpt, which are visually realized as clickable links in visXcerpt.

Figure 2.35 shows a simple program written in Xcerpt (in its abbreviated syntax, and not the XML format here) and its rendering in visXcerpt, clearly showing their direct relation.

The program itself consists of a rule with a query and a construct part. In the query part, the "FROM" part in the textual and the right part in the visual program, a file named "proceedings04.xml" is read in. It is then queried for an element "proceedings04" that contains an element "papers" which itself consists of "paper" elements. If such elements are found, the program adds the values of their "author" and "title" attributes to the variables "Author" and "Title", which are both defined as lists. In the construct part of the program, a new XML element "results" is created. The "all" operator then creates child elements for it by taking each title, author pair from the list variables, and constructing a new element "result", holding the respective values, for each pair.

[20]note that visXcerpt programs can not be executed directly, since they contain additional information required for their visual display. Before execution, they are transformed to valid Xcerpt through a XSLT program

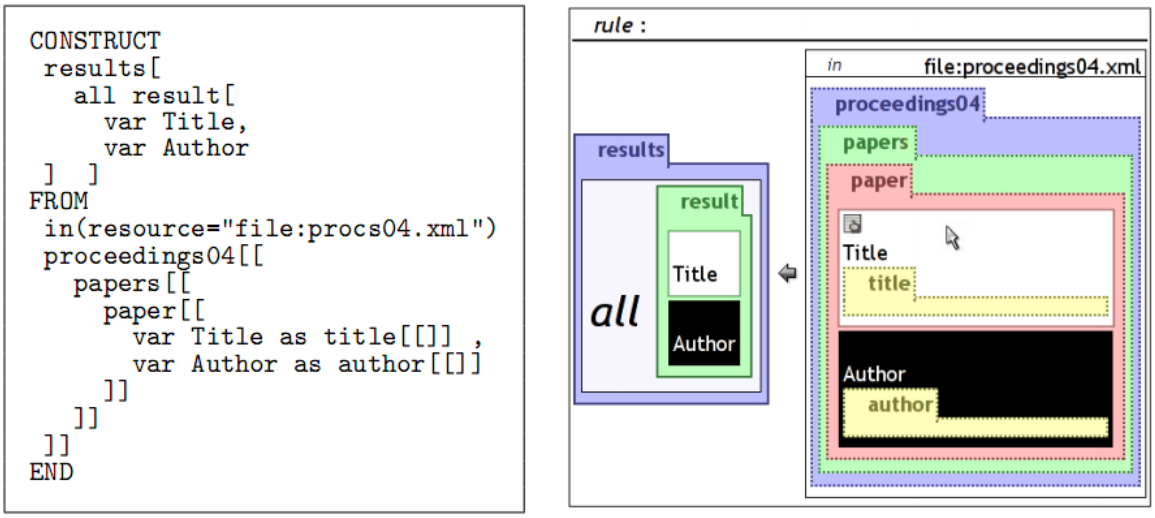


Figure 2.35: A Xcerpt program and its rendering in visXcerpt

2.4.2.6 NITELIGHT

In recent years, research is moving towards the Semantic Web, and query languages for it. The KiWi system (which will be introduced in chapter 3.1) is one example of a Semantic Web application, with KWQL (see chapter 3.2) being a specialized query language for it. The visKWQL language introduced in this thesis can thus be seen as a visual semantic query language.

A more general semantic web query language is SPARQL (see chapter 3.1.4.1).

SPARQL is a relatively new language, however, a diagram-based visual language that allows the graphical construction of SPARQL queries has already been implemented.

The visual query system, consisting of the vSPARQL language and an interactive editing environment for ontology navigation, is called NITELIGHT [56].

The motivation behind NITELIGHT was to create a tool that exploits the full power and expressivity of the textual target query language SPARQL and employs user interaction capabilities that allow efficient design of accurate queries.

Unlike in other visual query systems, in NITELIGHT expressivity is more important than ease of use, and NITELIGHT is therefore intended for users who have previous experiences with SPARQL, and is unsuited for those who have no familiarity with it, since it has a close correspondence between its graphical notations and SPARQL constructs, which might not be as easy to grasp for new users as for example the relatively simple and straightforward structure of XML and the query languages for it.

SPARQL, and therefore NITELIGHT, are working on the RDF data model [57], wherefore their queries are based on triple patterns. vSPARQL represents these triples with graphs.

The graph nodes represent a triple's object and subject elements, which can be URIs, literal values or variables. The node's shape and color are used to express the element's type. Object and subject nodes are connected with a labeled line, representing either a predicate or a query variable.

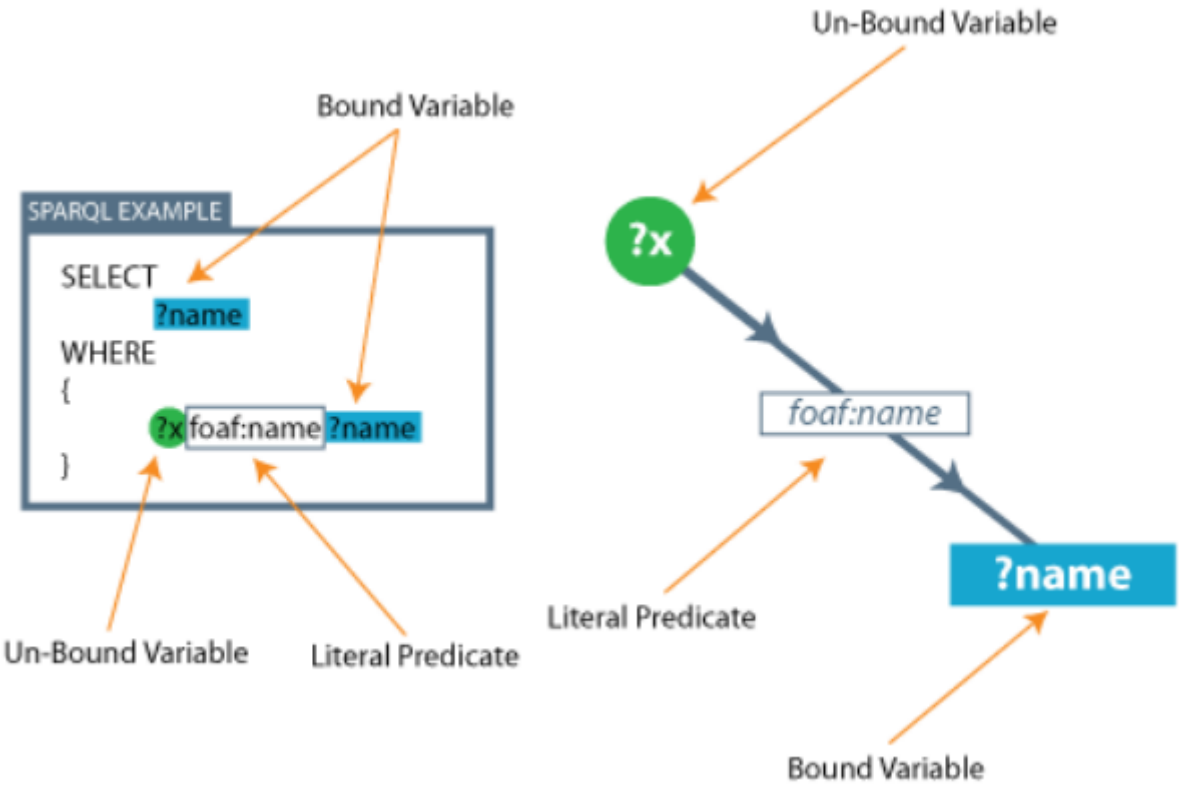


Figure 2.36: A triple pattern in SPARQL (left) and NITELIGHT (right)

Figure 2.36 shows NITELIGHT's representation of a SPARQL triple pattern.

Multiple triple patters are represented by graphs with additional nodes and connections, connected by nodes representing shared variables or values. Ordering of variables and triples can be represented with numbers next to node labels. SPARQL's graph patterns, collections of triple patterns matched against the entire RDF graph, where variables are not shared, but are local to a graph pattern and could be bound to different values, are represented graphically by grouping triples within boxes. Different coloring of the boxes, and connections between them, allow the representation of optionality and unions (see Figure 2.37). Graph patterns can also be assigned to variables or literal values (URIs), to allow specification of a default RDF graph or the retrieval of a graph as part of a query.

Ordering of results is expressed through a directed arrowhead within a graph node, and text boxes below variable nodes allow the entry of numerical or regular expressions for result filtering.

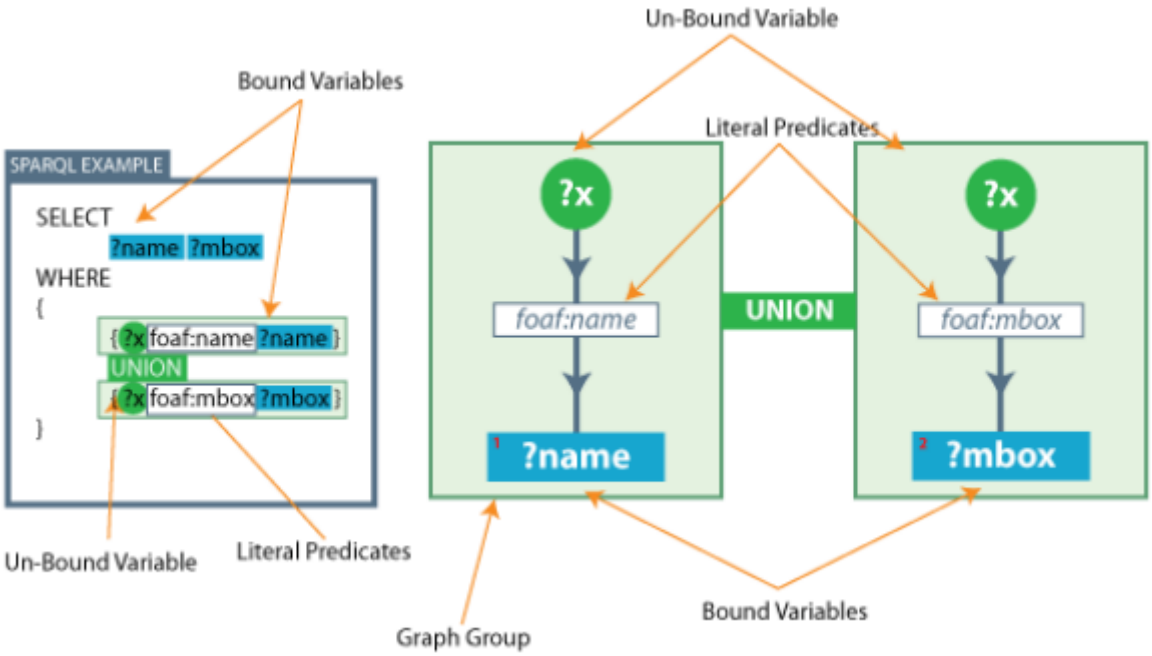


Figure 2.37: Union graph patterns in NITELIGHT

The NITELIGHT prototype consists of four parts:

- the *Query Design Canvas*: a graphical query editor, which allows the user to move elements freely, edit them through context menus, and zoom in and out
- the *SPARQL Syntax Viewer*: a dynamically updating display of the textual SPARQL query corresponding to the graphical query, similar to the output view of the KWQL Query Builder (see chapter 3.3.3.1)
- the *Ontology Browser*: a text-column based display of the source ontologies and their classes
- the *Query Results Viewer*: a table-based display of query results

2.4.2.7 RDF-GL

RDF-GL [58] is another visual query language for the semantic web, or more precisely RDF data, based on the textual SPARQL language.

It employs a visualization that is, although also graph-based, very different from the previously introduced NITELIGHT.

Queries in RDF-GL are represented with boxes, circles, and arrows.

Orange boxes, called result boxes, contain information about a query's execution, for example query type and result order. Pink boxes represent the subject and object elements of a SPARQL triple. Green boxes represent filtered subjects or objects.

Blue circles can be connected to elements to represent union, while purple circle represent optionality.

Black arrows depict SPARQL predicates, while gray arrows represent optional statements. Yellow and red arrows are used to indicate the first and second part of a union.

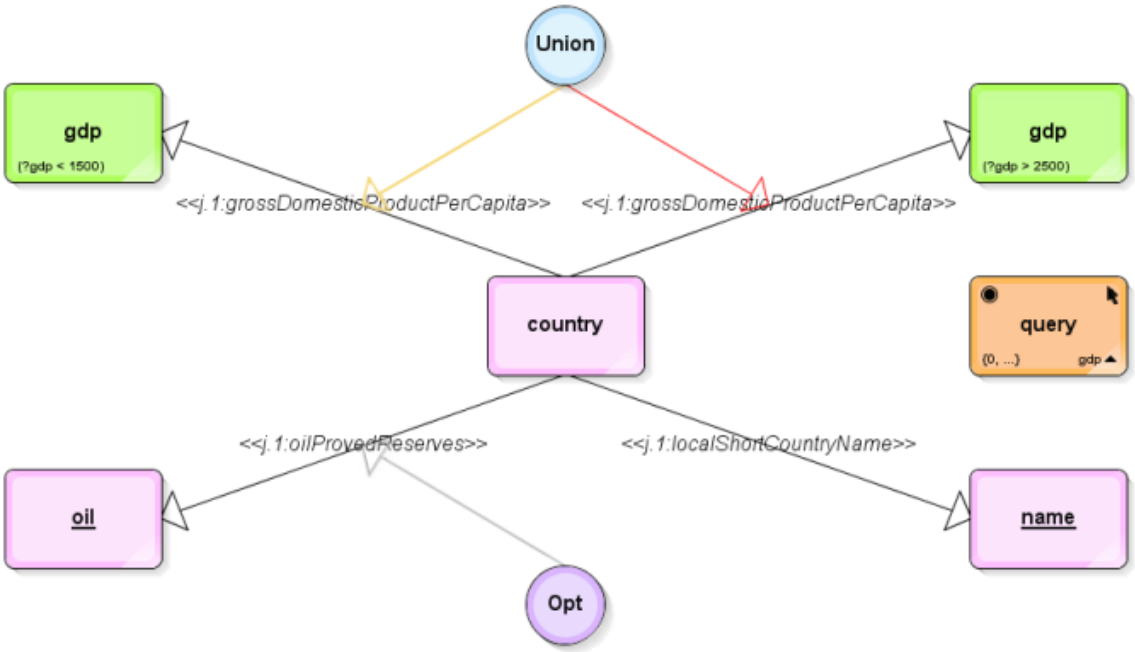


Figure 2.38: A RDF-GL query

Figure 2.38 shows a simple query in RDF-GL that uses all the different possible elements. Note that the lines under oil and name indicate that these variables will be included in the output. The orange box also shows the attributes that boxes can have, represented by a number of symbols in the corners of the box, which indicate whether all or only distinct results should be displayed, the type of the query, the result order, and the range of results. The green boxes contain a filter criterion.

The example query returns the name (and, if present, the oil supply) of every country with a gross domestic product per capita less than 1500$ or greater than 2500$.

2.4.3 Hybrid Systems

2.4.3.1 PESTO

As mentioned earlier, some visual query system take a hybrid approach, using a combination of form-based, diagram-based and icon-based[21] approaches.

Hybrid systems generally come in one of two different flavors.

The first class of hybrid systems employ two or more different visual formalisms together to visualize data and queries. Examples for this class include Pasta-3 [59], a visual query system operating on the entity-relationship schema of a prolog database, OdeView [60] to query object oriented databases, and PESTO [61], an integrated browser and query system for object databases.

PESTO employs a “query-in-place” paradigm that extends navigational browsing facilities by integrating query capabilities into them. This is accomplished by allowing the user to browse the database through navigational links, and specify filters in place, on any level of

[21] purely icon-based visual languages are almost non-existent due to the difficulty of reaching a practically usable level of expressivity with them, which is why there are no examples for such systems introduced here

nested sets, and with arbitrary complex predicates including joins. Furthermore, an iterative query refinement throughout a query session is possible.

The chosen visualization combines form-based and diagram-based approaches. Objects are shown as forms, with text- and other fields that display object attribute values, allow the user to edit them, and allow the setting of query predicates. The forms also contain a number of buttons, for example to switch between the different instances of the object, to follow references, or to invoke methods. The forms themselves are the nodes of a graph, with connections between them indicating object relationships.

PESTO supports basic features such as selection, value-based joins, negations, disjunctions, as well as more complex queries involving universal quantification and structural predicates. However, relational joins (which would produce new relations), projections and unions are not supported.

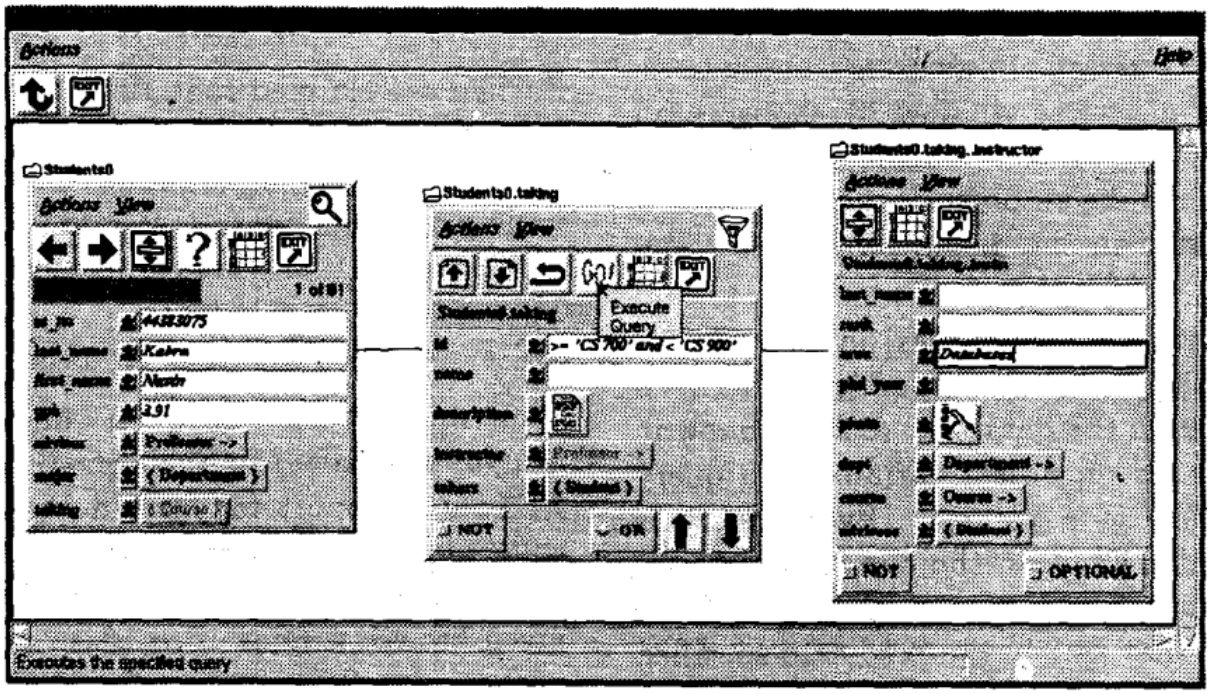


Figure 2.39: A Screenshot from PESTO

Figure 2.39 shows a screenshot[22] from PESTO, showing students and the courses they are taking which are taught by database professors.

2.4.3.2 SKI

SKI (Semantics-Knowledgeable Interface) [62] is a visual query system that sits on top of a specialized semantic database implementation.

The user is presented with a graphical representation of semantic schemas and uses semantically-motivated operators to create a *session view*.

There are three classes of such operators, allowing the user to explore semantic relationships by examining attributes, types and constraints, to explore component relationships by

[22]unfortunately, implementations of PESTO (and the other mentioned hybrid systems) are not available, so that no clear screenshots are possible

examining paths, and to view the *update environment* of a component with respect to some update operation, in which case the system displays the components likely to be affected by the operation.

SKI's visualization method is a combination of form-based and diagram-based approaches, used together.

The screen is partitioned into a variable number of horizontal stripes which represent the structure of the semantic schemas.

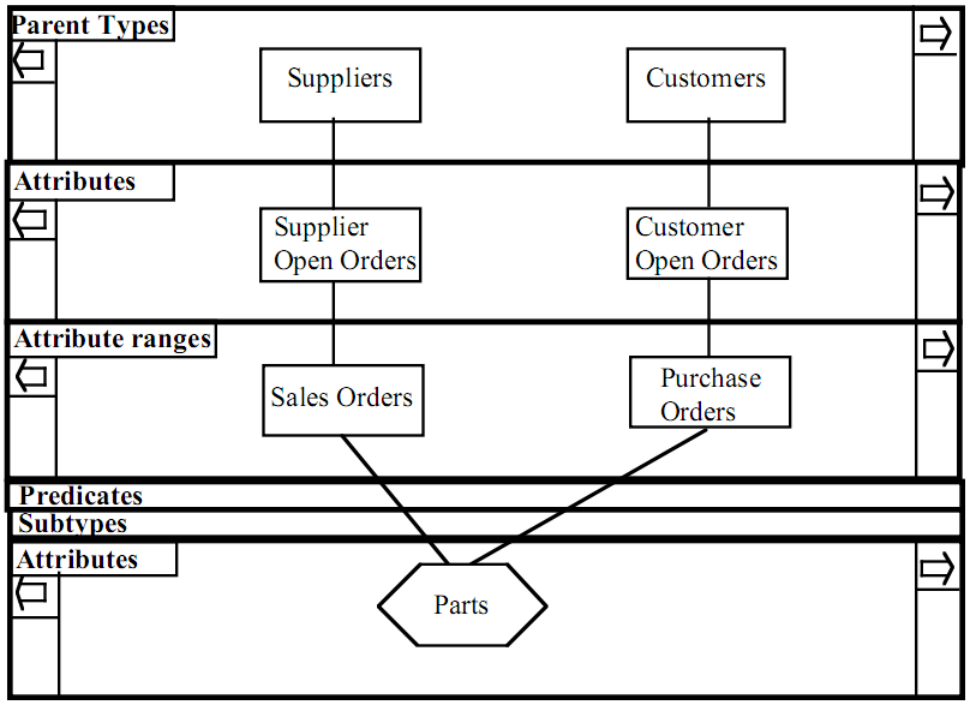


Figure 2.40: Visualization in SKI (from [9])

Figure 2.40 shows how SKI visualizes a relationship between two components (Suppliers and Customers).

The top stripe contains parent types, the second stripe their attributes. Further below follow attribute ranges (those can be either complex components like in the example, or terminals like Real or String), predicates and subtypes. The example demonstrates *information hiding* in SKI, which allows certain stripes to be collapsed (here the stripes for predicates and subtypes) to hide their contents. Also note the arrows on the sides of the stripes, indicating that horizontal scrolling is possible there, to display more components.

The bottom-most stripe in the example shows the attributes of the range types in the third stripe.

The example image is the result of a session, in which the user first selected the Suppliers and Customers components, and then used the *How Related* operator, which finds the shortest connection between the two components in the schema. The path between them connects them through intermediate types and attributes, and forms a graph this way.

Predicates in SKI can be entered into a form-based skeleton graph node in the predicate stripe.

Finally, query results, like attribute values for all components that fulfill some predicate, are returned through a form-based data view window.

2.4.3.3 IVQS

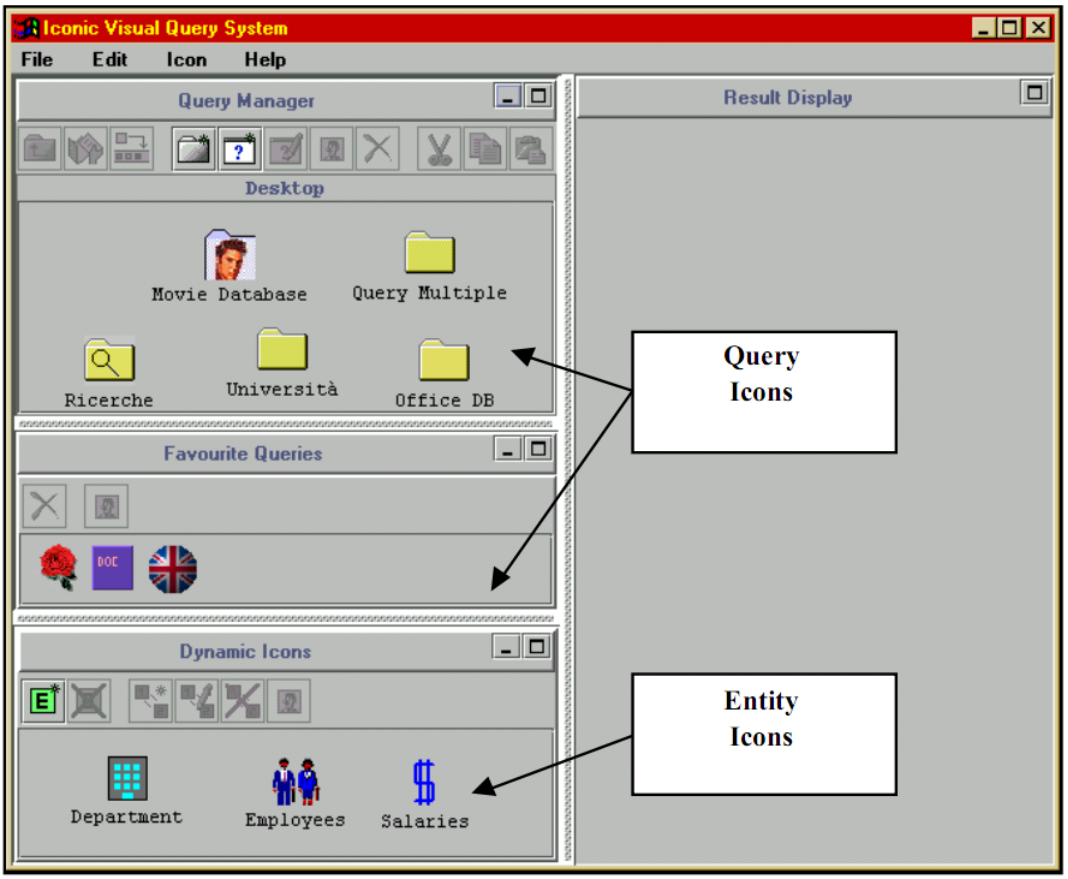


Figure 2.41: The IVQS system from the end user's view

The second class of hybrid systems does not combine different paradigms in one visualization scheme, but employs different visualization schemes based on different paradigms, and lets the user interact with one of them at a time.

Due to the inherent problems of such systems, basically requiring two or more individual visual query systems to do the same thing, they are almost non-existent. One exception is IVQS (Iconic Visual Query System) [63], which includes a form-based and an icon-based visualization.

One feature, apart from its hybrid visualization scheme, that sets IVQS apart from previously mentioned database query systems, is that it sits not on top of one database management system and allows querying one database, but that it allows the querying of multiple remote databases, relational multimedia databases to be exact. It even supports querying of multiple, and different, databases within one query, for example to facilitate a join operation between a table in an Oracle database and a table in an Access database. This way, IVQS can act as a visual tool for distributed data warehousing.

IVQS uses two different visual approaches for two different kinds of users.

Expert users, acting as administrators, are provided with a form-based SQL query editor, which allows query creation through selection of query elements (databases, tables, attributes, sorting keys and conditions) from lists. The expert users can thus create and

modify queries, entities, and a result layout, which is available either as a table or as a form variant.

End users are provided with an iconic environment (see Figure 2.41).

There are two kinds of icons in IVQS: *Query Icons* represent entire queries, each specified by an expert user through the form-based SQL editor.

Query Icons can be grouped into folders, and assigned to a special *Favourite Queries* area of the user interface. A simple double click executes a query, and displays the result in the way specified by the expert user who created the query.

Entity Icons, also created and specified by the expert user, represent database entities, that is relations. To each pair of entity icons a query can be associated. If the end user drags and drops one entity icon on top of another, the associated query, if available, is executed.

It can be seen that IVQS uses two different visualization approaches to achieve two different ends. Administrators can create queries with full SQL expressivity in a form-based editor. End users, however, are, through the iconic interface, restricted to predefined queries. They don't have to worry about knowing anything about the database schema or query formulation and need not do more than click or drag an icon to execute queries relevant for their work, but if their requirements change they have to let someone else create a new query.

Chapter 3

A Visual Query Language for KiWi

This chapter introduces visKWQL and the KWQL Query Builder, a visual query language and its accompanying editor.

First, however, the following sections give an overview of KiWi, the semantic wiki that will be queried, and KWQL, the textual query language on which visKWQL is based.

3.1 KiWi

> *Knowledge Management + Wiki Philosophy + Semantic Web = KiWi*
>
> *Sharing is Power*

KiWi (Knowledge in a Wiki)[1] [1] is a *Semantic Wiki* focused on knowledge management, aiming to combine the traditional human-readable wiki with computer-readable data and meta-data, to allow automatic reasoning and other features to enhance both the data and the user experience.

It builds upon previous efforts in the design of Semantic Wikis, like IkeWiki [64].

3.1.1 The KiWi Vision

KiWi aims, "to combine the wiki method of collaborative content creation with the technologies of the Semantic Web to bring knowledge management to the next level" [1].

The wiki method, or rather philosophy, consists of a number of principles for content creation:

Anyone can edit: Anyone can easily contribute to the content, without much restrictions or a strict hierarchy

Ease of use: Contributing content does not require more than basic word processing software skills

Linkable content: Anyone can create links between words, so that a creation of semantic relations is possible

[1] The KiWi project page can be accessed under http://www.kiwi-project.eu/ . Please also note that, as of writing this, the KiWi project has only completed roughly half of its development cycle, so this chapter is more an introduction to what is planned for the KiWi system, than to the system itself

Versioning support: No content is ever lost. When a page is changed, the previous version is stored and still accessible

Media support: Content is not restricted to a specific format like text

It should be mentioned here, that while most people think of Wikipedia when they hear wiki, that wikis are very versatile, and can be used in any place where collaborative knowledge management is useful, be it for professional knowledge management within a company, or a specialized social website.

KiWi itself already has some example use cases which demonstrate its versatility. One is the use of KiWi as a platform for social software applications in the form of a website for photo stories. Two other use cases employ KiWi as a knowledge management system within a company, one for IT project management, the other for software development management[2].

The second area, where KiWi is taking inspiration from, is the Semantic Web.

The Semantic Web aims to provide a flexible set of structures, which are applicable to different types of content. These structures are not seen as an obstacle, but as a useful addition to content, which support the user. The structures, and relationships between them, are not only inherent in the data, but can be made explicit, so that a software system can understand them and reason on the data.

Those structures, however, are not enforced. The Semantic Web is semi-structured, meaning it does not enforce structures, but only checks if available data is consistent with a model of knowledge representation, called an Ontology[3].

It also moves away from the *closed world assumption* of traditional database systems, where something that is not seen does not exist, towards an *open world assumption*, where absence of something does not equal non-existence.

KiWi tries to apply the Semantic Web's semi-structured nature to the wiki principles, to prevent chaos and give the content more meaning, to enable more efficient searching and navigation, automatic reasoning, and to enhance the level of social interaction between the system's users, by including for example FOAF (Friend-Of-A-Friend) data.

Like all social software, KiWi is a user-centered system. It considers five aspects:

Role: Each user can accept a role, corresponding for example to the user's position within a company. A user's role includes the knowledge he has, as well as his tasks and responsibilities

Domain: KiWi differentiates between different domains, or fields of knowledge, including concepts and relationships between the concepts that make up a domain

Context: KiWi is able to reflect some contextual aspects of tasks, like "What is it about?" or "Who is going to use the results?"

Experience: Different users have different levels of experience, and KiWi tries to accommodate them, by not requiring too much from inexperienced users, and not requiring too little from experienced ones

[2] see http://www.kiwi-project.eu/index.php/use-cases for more information about the use cases

[3] a formal representation of a set of concepts and relationships between those concepts

Preference: Users are allowed to select their preferred style of presentation and interaction with the system, to enhance their willingness to work with it. Options include for example background colors or diagram formats

Through this combination of methodologies, KiWi aims to enable "*Knowledge Management 3.0*", where knowledge is created because knowledge is shared. It will use versioning to promote trust and confidence in the system, will be open and flexible for all kinds of content, will reflect and enhance community relationships, and will be able to be custom-tailored towards the knowledge management needs of a particular organization or community.

KiWi will also include a highly customizable user interface and support for advanced concepts such as automatic information extraction and rule-based reasoning, and of course include a powerful query language, KWQL, which will be introduced in chapter 3.2.

3.1.2 The KiWi System

From a technology perspective, the wiki philosophy does not only apply to wikis themselves, but also to other social software like blogs, photo sharing sites, social networking platforms, and more [1].

Based on this insight, it is KiWi's aim to *break system boundaries* between these kinds of services and not only provide a wiki functionality, but to become a reusable platform for all kinds of social software services.

The core functionality, based on the wiki philosophy and required by most social software systems, that KiWi will provide, consists of:

- representation of many different content formats
- easy linking
- versioning and change tracking
- annotation of content with meta-data
- rating and commenting of content by users
- integration of content using machine-readable connections
- user management, user profiles and social networking
- personalization and context adaption
- reputation and incentive system
- information extraction
- reasoning and recommendation system

KiWi will not only be a framework for different social software services, but it will allow the creation of different services in the same instance, using the same content, to allow, for example, the creation of a photo-sharing site, a photo-story site and a photography learning blog, which all use the same photos, no matter on which of the sites a particular photo was uploaded.

Furthermore, KiWi treats "widgets", functionality and user interface components, like any other content data, and applies the "anyone can edit" principle to them, so that everyone can become a developer of the platform itself, by creating, editing, combining, and sharing system functionality through widgets.

In addition to *breaking system boundaries*, KiWi will also *break information boundaries*, by introducing content integration that goes beyond mere linking, making not only content, like an image, available through a link, but also its available meta-data, like image properties and annotations.

The user will be supported through a flexible personalization module in KiWi, that will make use of *roles*.

For example, a user may have a "work" and a "home" role. When at work, the system will display news from the company website on the wiki homepage, while when the user is at home, and assumes the "home" role, the system will display him content he is more interested in personally, for instance recently uploaded travel images.

Finally, KiWi is planned from the beginning to be applicable in corporate settings, which are quite different from community settings in some regards. In a corporate setting, usually the number of contributors is low, leading to little and poorly maintained content, there is no intrinsic motivation to create and update content, especially because the motivation to share knowledge is often low, since many people hold on to the "knowledge is power" notion and are reluctant to share. Furthermore, textual content is insufficient for corporate needs, since it is for example often necessary to share layouted text or documents in office formats. Also, unlike in community settings, protection of corporate knowledge is a must, making permission management and support for different access rights necessary. Finally, corporate settings may need workflow support.

KiWi will accommodate all those needs, by including workflow support, permission management, support for rich text editing and many different data formats, including the possibility to later include not-yet existing formats through external loaders. To encourage sharing and participation of users, the SunSpace community equity system [65] will be included in KiWi. This system will calculate scores (equities) for all content items and users by measuring certain aspects, like quality and quantity of contributed content from a user, to encourage users to participate more to heighten their score.

3.1.3 KiWi Usage Concepts

KiWi consists of a number of core concepts [1] which are important here, because these include the resources that will be queried through KWQL and visKWQL (also see chapter 3.3.3.2):

Content Items are the basic units of information in KiWi. By default, all content items are wiki pages. They are composable documents that can be viewed, edited and annotated by the user. A content item has both a machine readable symbolic and a human readable textual or multimedia representation. Each content item is uniquely addressable through an URI (Unique Resource Identifier). Content item composition is possible, by including other content items within one content item (for example including an image, which is a content item in itself, in a textual wiki page). Content items may

consist of overlapping *Content Item Fragments*, which also have URIs, and can be individually annotated. Content Items can also include *Links* to other Content Items or Fragments. Special wiki data, like user profiles, roles, layouts, widgets, etc. are also represented as content items.

Tags serve a number of purposes in KiWi, for example grouping content items, or associating them with certain topics. *Explicit tags* are annotations created by the user, and can be seen as relations between two content items. *Implicit tags* are created automatically by the system, for example through information extraction from text or through reasoning on existing tags.

RDF Annotations: RDF (Resource Description Framework [57]) is the format in which KiWi internally stores meta-data and annotations. It is a common format, and thus allows the integration of KiWi with existing Semantic Web applications.

Users play a central role in KiWi. Each user profile has an URI and at least some core data like name, login and password, and is represented as a content item. Social networking between users is supported through the modeling (and RDF export) of social relations and an implementation of the Google OpenSocial API[4] to make integration with other social networking applications and Semantic Web applications possible. As mentioned earlier, KiWi includes the SunSpace system for user ratings.

Roles were already mentioned in the last section, and allow one user to have multiple models for interaction with the system.

Context consists of the current user, the user's current role, and the currently viewed content item. It determines what is displayed, how it is displayed, and the user's interaction possibilities.

Revisions are logical updates to the system's knowledge base. Unlike in normal wikis, revisions in KiWi are more complex, since a simple update can affect multiple content items (because of content item composition and the advanced linking between them). Revisions are thus attached to all content items involved in an update. Users have full access to a content item's revision history.

Two advanced concepts of KiWi are rules and structured tags.

Structured tags are complex tags, consisting of multiple simple (atomic) tags, to make more sophisticated tagging possible, for example one could tag a hotel he liked with the structured tag *hotel(comfortable)* instead of just the atomic tag *hotel*.

Rules are part of the KWQL (and visKWQL) query language, and allow the construction of new content items. In KiWi, rules are not seen as mechanisms to only run once, but as special content themselves. They are thus represented by special content items, including the rule code and an optional explanation. Rules can be included in other content items, to be executed whenever that content item changes. This way it is possible to, for example, write a rule that will automatically create a table of contents for a wiki page, and include it in the page. Whenever a user edits the page, the system then executes the rule and automatically creates and includes an updated table of contents on that page.

[4] http://code.google.com/apis/opensocial/

3.1.4 Querying KiWi

KiWi has its own textual query language, KWQL (see chapter 3.2), which is specialized to query KiWi's data model. However, internally KiWi stores its data as RDF triples, which make the integration with other Semantic Web software possible, including query languages for the Semantic Web.

There is quite a number of query languages for RDF and the Semantic Web[66, 67], the most notorious of which is SPARQL [68], which will be briefly introduced here, since KWQL itself supports the inclusion of SPARQL queries.

3.1.4.1 SPARQL

The SPARQL query language (the name is a recursive acronym for SPARQL Protocol and RDF Query Language) [68] is a standardized query language[5] for the Semantic Web, or more specifically for RDF (Resource Description Framework) [57].

RDF data comes in the form of arbitrary (though usually labeled, directed, multi-) graphs, where all properties are optional and multi-valued. The graphs consist of RDF statements, which are subject-predicate-object expressions (also called RDF triples).

The subject is a resource, which either has an unique identifier (URI) or is anonymous (a blank graph node). The predicate is also an URI indicating a resource, representing a relationship between subject and object, and the object may be an URI or blank node like the subject, or a string literal.

In most applications of RDF, the URIs point to actual data on the web. The RDF format, however, is more general, and an URI can practically represent any possible kind of resource.

SPARQL is the predominant query language for RDF. Its syntax is somewhat rooted in SQL, with queries including SELECT and WHERE clauses, and makes use of triple patterns, which correspond to the RDF triples, but allow the use of variables instead of URIs.

For a simplistic example, assume we query data consisting only of the single RDF triple

```
<http://example.org/book/book1>
<http://purl.org/dc/elements/1.1/title>
"SPARQL Tutorial" .
```

where the subject is an URI pointing to the http URL of a book, the predicate points to a web resource representing the property "title", and the object is the string "SPARQL Tutorial", so that the triple represents the statement that the title of book1 is "SPARQL Tutorial".

Now assume we want to query this data, to find out the title of the book. The SPARQL query reads

```
SELECT ?title
WHERE
{
```

[5] sharing its name are the SPARQL Protocol for RDF, a remote protocol for issuing SPARQL queries and receiving the results, and the SPARQL Query Results XML Format, a XML format for representing the results of specific SPARQL queries

```
        <http://example.org/book/book1>
        <http://purl.org/dc/elements/1.1/title> ?title .
    }
```

The query thus selects (and returns) the variable "?title"[6], whose value will be the object of the RDF triple with the URIs for book1 and the title property as its subject and predicate.

In addition to querying, SPARQL also has construction capabilities, which allow the creation of new RDF graphs.

Consider for example the query

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
PREFIX org:    <http://example.com/ns#>
CONSTRUCT
{
    ?x foaf:name ?name
}
WHERE
{
    ?x org:employeeName ?name
}
```

First, the query demonstrates the use of the PREFIX statement, which makes queries more readable, by specifying URIs before the actual query, and assigning names to them, which can then be used in their place within triple patterns. So in the example, foaf:name actually points to http://xmlns/foaf/0.1/name.

There is a number of additional ways to make queries shorter, for example if triples share subject and predicate, objects may be separated by commas.

The query also shows the use of CONSTRUCT instead of SELECT, which simply creates a new triple (in this case; a CONSTRUCT term can create arbitrary graphs consisting of multiple triples), using variables which were bound in the WHERE clause. In this case, the query turns a graph with employee names into a graph with FOAF (Friend Of A Friend) names.

Similar to SQL, SPARQL queries may also contain a FROM clause, which specifies a default graph to query. Querying named graphs is supported, too.

SPARQL also supports the use of filters to restrict results. These can be regular expressions on strings, arithmetic expressions, and additional constraints like type filtering. Triple patterns can also be indicated as being optional, and combined in unions. The query results can be ordered, their number limited, and a range of results to be returned can also be specified.

In addition to the SELECT and CONSTRUCT queries, SPARQL provides two more query types.

[6] all variables are indicated by a "?" or "$" prefix

ASK queries can be used to test whether a query has a solution, and return only a single Boolean, yes or no, and no further information about the query solutions.

DESCRIBE queries take the resources from a query solution, and optionally directly named resources, and return a graph that describes those resources, which the SPARQL query processor assembles from all available data.

3.2 KWQL

KWQL (KiWi Query Language, spoken "quickel") [2, 69] is a query language specifically developed[7] for KiWi.

While KiWi has an underlying RDF triple store, and can thus be queried through SPARQL, SPARQL conflicts with the KiWi vision. KiWi aims to provide a framework for use in many different scenarios, in the enterprise as well as in the community, giving it a wide spectrum of potential users. SPARQL, however, is a language that is not useable without both an understanding of the RDF data format, and its own (more or less complicated) syntax, both of which the average KiWi (or social software) user is unlikely to have. And requiring the users to learn SPARQL to be able to find the content they are looking for would be problematic for the success of the KiWi project.

KWQL attempts to overcome this problem, by combining powerful construction capabilities, the expressive power of a *query language* like SPARQL, and the ease of use of a *search language*, like it is employed in an Internet search machine, making it possible to both write complex rules and single keyword search queries. It holds additional advantages over SPARQL, like easy support for negation, and very tight integration with KiWi. While SPARQL is a query language for general RDF data, KWQL directly builds upon KiWi's conceptual data model.

KWQL can be described as a rule-based query language based on the label-keyword paradigm. It can query and construct full text, KiWi's document structure, annotations and meta-data.

The design goals that KWQL strives to fulfill are

Low Entry Barrier: a new user should have to know nothing about either the query language or KiWi's underlying conceptual model, and still be able to formulate searches. The learning curve, once he gets familiar with the system, should be smooth.

Full Awareness of the Conceptual Model: the query language should make full use of the underlying conceptual model by incorporating its structure and properties.

Answer-closedness: the data returned from a query should adhere to the same data model as the queried data, so that query results can be queried again.

Monotonicity: as a query gets more complex, it should become more selective.

Construction: the language should support the creation of new "views" of the data through selection and construction.

[7]Please note that, as of writing this, KWQL has not been fully implemented, and the information in this chapter is therefore subject to possible small changes

Continuous Queries: queries (and constructions) should be able to be created once, and be automatically evaluated again when the data changes.

No Striving for Completeness: the language should focus on simplicity and ease of use instead of relational completeness.

Future possibilities for KWQL include *rule chaining*, the execution of rules one after the other, each using the result of the previous rule as input, and grouping and ranking of results.

KWQL holds true to KiWi's principle of being easy to use for inexperienced users, but giving experienced users a lot of power. Thus, the simplest kind of query in KWQL is just a single keyword, constituting a full text search over the whole wiki, just like the search function of a website or a search machine like Google.

For more sophisticated queries, KWQL makes use of KiWi's underlying conceptual model of resources and qualifiers.

The resources, which were already mentioned in chapter 3.1.3, that can be queried by KWQL are *content items* (which are abbreviated as *ci* in KWQL), *fragments*, *links*, and annotations (*tags*).

Qualifiers are the meta-data of resources, like *author*, *title*, *text*, or *URI*. For a complete listing of the resources and their accompanying qualifiers see chapter 3.3.3.2.

Queries involving structural information follow the pattern

`resource(qualifier1:value1 qualifier2:value2 ...)`

First a resource identifier, then a list of label-keyword pairs.

If no qualifier is specified, then a full text search over every qualifier is assumed.

Thus, the single keyword query `XML` would be identical to `(text:XML OR title:XML OR author:XML OR ...)` and return all content with some property that includes the string "XML". Note that in this case, also those results which just contain "XML" within a larger string will be returned. If the value of some property should be "XML" alone, the value must be put into quotation marks in the query, e.g. `"XML"`.

A search for multiple keywords in which their order is important can be accomplished by putting the keywords into parentheses in the order they should appear in the text.

Resource identifiers can be nested where the conceptual model allows it, since content items can contain fragments and links, and content items, fragments and links can contain tags. For example, to find all content items that have a tag that was created by Mary, one could use the query `ci(tag(author:"Mary"))`. Please see Figure 3.21 in chapter 3.3.3.2 for a table listing the possible qualifiers and sub-resources for all resources.

Queries can also include several operators which are self-explanatory: conjunction (*AND*), disjunction (*OR*) and negation (*NOT*).

If label-keyword pairs are listed within a resource and no operator is given, conjunction is implicitly assumed.

`tag(author:"Mary" name:"wiki")` is thus equal to `tag(author:"Mary" AND name:"wiki")`, and would return all "wiki" tags written by Mary.

`ci(author:(NOT "Mary"))` would return all content items that were not written by Mary.

Operators can be used to form arbitrarily complex patterns with parentheses, like

```
ci(((text:XML OR text:wiki) AND author:"Mary")
OR (NOT(author:"Mary" AND text:KiWi)))
```

to search for all content items that were written by Mary and include “XML” or “wiki” in the text, or were not written by Mary and also include the text “KiWi”.

KWQL is, however, not only able to query KiWi, but also supports sophisticated construction capabilities based on rules.

A rule has the general form *construction-part @ query-part*. Instead of returning a content item that fits the query, it constructs a new content item based on the query.

The query part is a query as shown before. However, when used in a rule, a query part can include additional elements. The construction part is a content item description following the same pattern as a query, but also including some additional elements.

Variables, indicated by a keyword preceded by a “$” symbol, can be used in rules to “transfer” data from the query to the construction part. For example, the rule

```
ci(title:"Mary's work" author:"Mary" text:$X) @
ci(author:"Mary" text:$X)
```

would search for a content item written by Mary and create a new content item with the same text and the title “Mary’s work”.

Bindings are a combination of query keyword and variable binding. The rule

```
ci(title:$X) @ ci(title:XML->$X)
```

would search a content item that contains “XML” in the title, and then create a new content item with the same title, no matter how the whole title string looks like.

The query part in a rule can also contain optional parts. For example,

```
ci(title:$T text:$X) @ ci(title:XML->$T OPTIONAL ("empty") text:$X)
```

will search for a content item that contains “XML” in its title. When the content item has a text, it will be assigned to the variable X. If it has an empty text, the optional’s default value, the string “empty” in this case, will be used as the new content item’s text value instead.

Finally, there are some additional operators that may be used in the construction part in combination with variables.

`COUNT($X) @ ci(title:XML->$X)` will search all content items with “XML” in the title, and return the number of found items. KWQL assigns all found content item titles to the variable X, and the COUNT operator extracts the number of assigned items.

The ALL and SOME operators allow the extraction of all or some of the bindings to a variable. The rule

```
ci(title:Contents text:($A "-" ALL($T,","))) @
ci(title:$T author:$A)
```

shows the use of the ALL operator. Its first argument is the variable from which it extracts values, the second argument is a string that is used to delimit the values. SOME works similar, following the pattern SOME(x, variable, delimiter), where x is the number of values it will extract. The rule searches all content items and saves their authors and titles. It then constructs a new content item with the title “Contents”, and its text is constructed of an author name, followed by a “-” sign, followed by a list of all titles written by that author, with each title separated from the next by a comma.

Structured Tags are going to be supported in KWQL as well, through *tag groupings* and *tag characterization*s.

Tag grouping allows the grouping of values to form complex tags. Grouped values may

by either atomic tags or complex tags themselves. They are expressed simply as a comma-separated list of values for the name qualifier of a tag, for example `tag(name:(Munich, LMU))`.

Tag characterization allows the classification, or labeling, of a tag, for example `price:5$`. Nesting of tag groups and characterizations is possible, for example some place could be tagged by a geo-location, including exact latitude and longitude, as `geo:(lat:52.272135, lon:-1.595764)`.

Finally, as mentioned in the last chapter, since KiWi also saves its data in the RDF format, KWQL directly supports SPARQL queries. These can be posed in a query simply by adding `SPARQL:(some SPARQL query)` to the end of a KWQL query. SPARQL queries can also be used within a rule, and even the evaluation of variables from a SPARQL query in the construct part of a rule is possible.

The above examples show how KWQL has a smooth learning curve, beginning with single-keyword queries for the beginning user and slowly getting more complex and powerful with the introduction of resources, qualifiers, operators, and finally rules. While beginners can easily search for content relevant to some keywords they are interested in, intermediate users can use the language to include structural or meta-data constraints in their queries, and sophisticated users can use complex rules to create new content.

Rules, especially continuous rules that are posed once and re-evaluated by the system every time the content changes, allow automatic content creation ranging from the creation of tables of content for single wiki pages, to the creation of new pages collecting links to all pages relevant to some topic, and much more, making KWQL a powerful tool for KiWi.

3.3 visKWQL and the KiWi Query Builder

This chapter introduces visKWQL, the visual rendering of the textual KWQL language, and the KiWi Query Builder (KQB), the visual editor for visKWQL.

The following sections give an overview of the motivation behind visKWQL, its design goals and how they were realized, a detailed manual for the language and its editor, and a discussion of the implementation.

3.3.1 Motivation and Design Goals

The motivation for visKWQL was to create a visual rendering of the keyword-based KWQL query language, to be used in KiWi as an alternative of or extension to the textual language, especially suited for users unfamiliar with query languages.

To the best knowledge of the author, there are no other visual keyword-based query languages yet, so visKWQL is a first attempt at visually implementing a language with the unique qualities of a keyword-based query language, especially the low entry barrier.

The common aim of all visual query languages is to make querying easier, to accommodate layman users who have no prior experience with query languages and no desire to learn a complex textual language, but still want or need to query data. Most visual languages are therefore designed with the goal in mind, to take away the need for learning a complicated

underlying textual query language, or trading expressivity for ease of use by abstracting a textual language to an easier to grasp, but less expressive visual language.

KWQL, however, was itself designed following the keyword-based approach, to provide a combination of ease of use and high expressivity, and is therefore better suited for layman users than pure query languages. Because of this ease of use, where queries can consist of just a single keyword, visKWQL was intended to stay close to the structure of KWQL queries, and can therefore be seen more as a visual rendering of the textual language than many other visual query languages, especially graph-based ones, where the visual query is often radically different from its textual counterpart.

The visual approach visKWQL takes is thus that of form-based visualization similar to the treemaps of VXT (see chapter 2.3.2) and the nested containers of visXcerpt (see chapter 2.4.2.5).

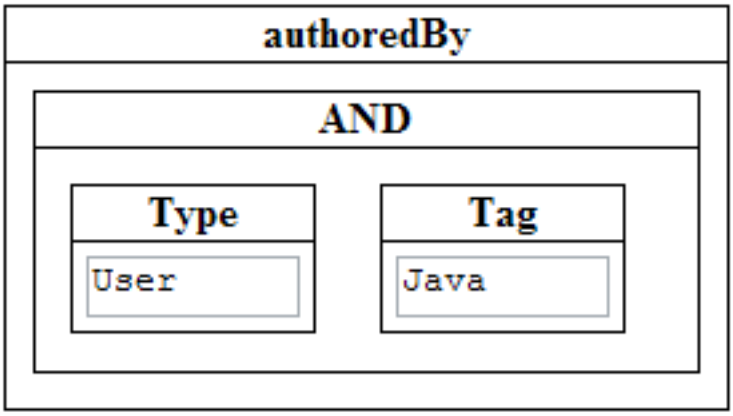


Figure 3.1: visKWQL query concept

Figure 3.1 shows a concept drawing of the intended query visualization scheme from the planning phase of visKWQL. The query corresponds to the KWQL query[8] `authoredBy ( Type:User AND Tag:Java )`. The closeness of the visual representation to the textual query can be seen. In fact, the textual query can directly be read from the visual representation by an experienced user.

While the query may be less of a visual abstraction than for example a query in a diagram-based language like G, the chosen form-based approach should be very easy to grasp for new users.

The chosen form-based approach was also influenced by the intended underlying technology of visKWQL.

Since KiWi is an internet application, accessed by users through their web browsers, the same must be true for the KiWi Query Builder. It should be usable by anyone with a web browser, and not require a user to install special software or browser plugins just to be able to query the wiki.

With the current browser technology, making use of DHTML (Dynamic Hypertext Markup Language, using HTML, Java Script and CSS to display web pages and handle user interaction), the realization of a graph-based query language would be problematic, since DHTML

[8]please note that this is not a valid KWQL query, since it was created when KWQL itself was still early in its planning phase

provides no support for features such as free shape drawing that would be needed. A form-based approach, making use of boxes and text, however, is realizable in DHTML.

In addition to an easily understandable graphical representation that requires little learning in itself, the aim of visKWQL and the KiWi Query Builder is to support the user during the query construction, wherever possible, to provide an incentive to use the visual editor.

One aspect of query construction is the actual editing process. The KQB should allow query editing in a straight-forward and easy way that most users will instantly understand.

The other aspect is that the KQB should aim to apply the concepts of user-centeredness of KiWi to the query construction process, providing both beginners and sophisticated users with useful help, making the swift construction of (at least syntactically) correct queries easier. To reach this goal, the user should be guided through the query construction with helpful hints and explanations, and the system should prevent (syntactical) errors in the queries where possible, and alert the user to them where prevention is not possible. It should also correct problems where possible, and inform the user of what and why it changed, to provide a continued learning experience.

Another goal for visKWQL, that is relatively rare in visual query languages, was *roundtripping*.

Roundtripping means that the user is not restricted to visually create and edit a query and then execute it, but that he can work on both the visual and the textual representation of a query, and see the changes reflected in the other representation. This way he can, for example, start with a simple textual query, add an element to it in the visual representation, and finally edit a value again in the textual representation before executing the query.

Overall, the design goals for visKWQL and the KQB were:

Full Expressiveness of KWQL: visKWQL should include all structures of the KWQL language and support the full grammar. Every possible KWQL query should be expressible in visKWQL

Visual Language as a Rendering of the Textual Language: visKWQL should stay close to the KWQL structure, which was created with ease of use and easiness to learn in mind

Implementation in DHTML: the KQB should be implemented in DHTML, with HTML and CSS for the presentation and Java Script for the program logic and user interaction. The visual representation must therefore be realizable in DHTML

Roudtripping: the user should be able to make changes in both the visual and the textual representation of the current query at all times, and see the changes reflected in the other. visKWQL should therefore both support immediate display of the textual query after a change in the visual query, and include a parser for KWQL, to be able to reflect changes in the textual language in the visual representation

User Support: the user should be supported in the query construction at all times. This includes both the language representation and editing features, as well as context-sensitive support to create correct queries, guiding the user so that even completely inexperienced users are able to immediately create useful queries without the need

of reading a manual, but also giving experienced users an incentive to use the visual language

Easy to Understand and Uniform Representation: KWQL elements (resources, qualifiers and operators) should all have an easy to understand representation, and all elements should be represented in an uniform way and not require the user to learn multiple different formalisms

Easy Query Editing with Drag & Drop: each computer user with a modern operating system is familiar with drag&drop techniques, which are therefore ideally suited for query editing and should be supported by the editor

Support for Information Hiding: complex visual queries can easily become larger than the screen, or at least so large that their comprehension gets complicated. The editor should therefore support some form of information hiding, to temporarily hide parts of a query, to make the part the user is working on clearer

The System Should be Self-explaining: the editor should provide tooltips for all elements, especially the KWQL structures that make up a query, to help the user build a query involving the correct KWQL resources and qualifiers, without the need to first read a manual explaining them

The System Should Guide the User: the user should never be unclear of what to do next. The system should provide some form of guidance or tips to the user

Error Prevention: the system should try to prevent or correct syntactic errors where possible. It should, for example, prevent the user from dragging a KWQL qualifier into a resource that can not have this qualifier. It should also visually show the users errors (like a string containing invalid characters) or problems (like an AND operator with only one argument) in the query, and, where possible, correct them. When the system corrects a problem (for example by removing the AND node) it should inform the user of this, to support him in learning correct practices

3.3.2 Language and Editor Features

The design goals detailed in the previous chapter have all been realized in visKWQL and the KQB. This chapter gives an overview of how they have been realized, and over additional features of the system.

3.3.2.1 Visual Formalism

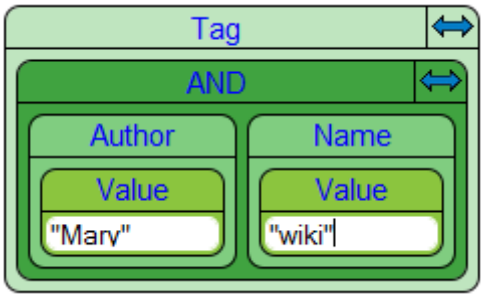


Figure 3.2: A visKWQL query

For the reasons given in the last section about its design goals, the chosen visual formalism for visKWQL is a form-based approach based on nested boxes.

Figure 3.2 shows an example of a query in the finished system. The visual query corresponds to the textual KWQL query `tag (author:"Mary" AND name:"wiki")`, which searches for "wiki" tags written by Mary.

The example shows how nested boxes form a query. Each box consists of two parts, a *Label*, and a *Body*. The label is the top part of a box and includes the name of the resource, qualifier or operator that the box represents, and it may also include a *Resize-Box*. The body is the main part of a box, and contains one or more child elements.

The example also shows three different types of boxes that form this particular query.

(a) Input-Box (b) (normal) Box (c) Fixed-width Box (d) Expandable Box

Figure 3.3: Different visKWQL box types

Figure 3.3 shows all the different kinds of boxes that are used in visKWQL to represent different KWQL elements:

Input Boxes: They are used for keywords, variables, and SPARQL input. Their body contains a textbox, in which the user can directly enter a value, like "Mary" in the example (see Fig. 3.3a).

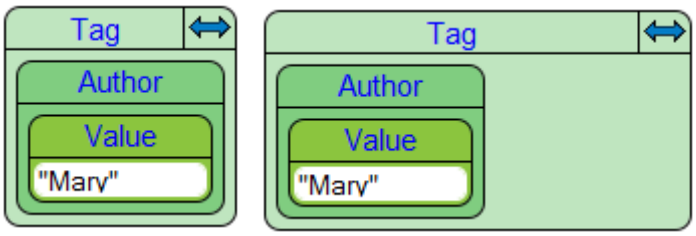
(Normal) Boxes: They have nothing special about themselves and only consist of a label[9] with the name of the box, and an empty body which can hold one child element. They are used for qualifiers and the unary NOT operator (see Fig. 3.3b).

Fixed-Width Boxes: These are used for elements that have a fixed number of children, like a Rule or the ALL and SOME operators. A rule for example must have two children,

[9]Please note that the orange color in this and the following images has nothing to do with the box type, but is a warning that the box is empty. This will be explained more in the section about user support through error and warning indicators

a construct and a query part. The box representing it thus has a wider body, indicating that it must hold more than one child element (see Fig. 3.3c).

Expandable Boxes: These boxes (see Fig. 3.3d) are used to represent KWQL items that can hold a variable number of children (KWQL resources and operators). To make queries visually more appealing and save screen estate, such a box usually only shows its actual child boxes and no free space like fixed-width boxes do. They contain a special resize-box in the label, indicated by a two-headed arrow which, when clicked, expands the box to show a free space to drop another child box in, or hide the free space when it is currently displayed. Figure 3.4 shows an expandable box before (Fig. 3.4a) and after (Fig. 3.4b) clicking the resize-box to show free space.



(a) Unexpanded (b) Expanded

Figure 3.4: An expandable box with a child box

One more of the initial design goals that the example in Figure 1.2 shows is the use of the same visual formalism of nested boxes for all different kinds of KWQL constructs. Resources, qualifiers and operators are all represented through boxes.

For resources, the visual inclusion of a child box corresponds to the addition of an expression within the parentheses in a KWQL query following the pattern `resource (child1 child2 ...)`.

For qualifiers, the visual child box corresponds to the expression after the colon in a query of the form `qualifier:value`.

For operators like AND and OR, the children of boxes representing these operators contain both the expressions preceding and following the operator in the textual query (e.g. `(child1 AND child2)`), similar to a pure graph-based visualization.

visKWQL also fulfills the design goal of having the same expressive power as KWQL, by including all KWQL resources, qualifiers, operators and other elements (like rules), and allowing the nesting in such a way that all possible KWQL queries can be represented.

Visualizing all KWQL constructs in an uniform way makes the language easier to learn and understand. However, giving the user some way to differentiate between different kinds of elements has been considered.

Many visual languages differentiate between types by using different shapes to represent them. This, however, conflicts with the goal of high similarity between elements (and would also be difficult to realize in DHTML). visKWQL thus uses colors to create easily noticeable, but not too distracting differences between different types of elements.

The prototype uses different tones of green, based on the green KiWi color-scheme.

Figure 3.29 shows a rule[10] that demonstrates the use of the four differently colored elements in visKWQL. Operators, like AND and ALL, are dark-green, resources, like ContentItem, a very light green and qualifiers, like Title, Text or Author a green tone in between. Other elements, like keywords or the rule, have an olive-green tone, while variables have a very bright green tone.

Together, the different colors both help to easily differentiate between different kinds of elements, and make the whole query visually more appealing than it would be if all elements used the same color.

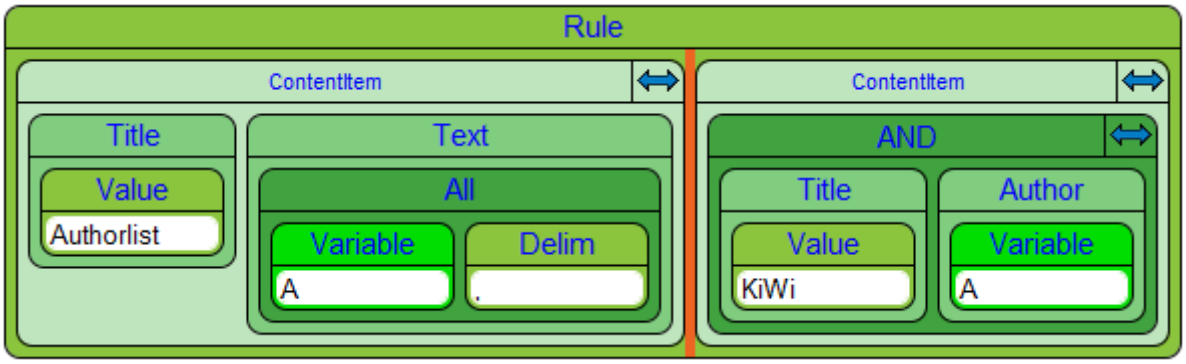


Figure 3.5: a visKWQL rule with all differently colored types of KWQL elements

3.3.2.2 Interaction Strategy

In the introduction to visual query languages (see chapter 2.4), the interaction strategy of a system was named as one of the important aspects of a visual query language. It includes both understanding the reality of interest, e.g. schema browsing, and query construction.

Since the conceptual schema of KiWi is constant, consisting of (nested) content items, which can include fragments and links and have annotations, schema browsing is unnecessary in the KWQL Query Builder. Similarly, database exploration or browsing in a database query language corresponds to simply browsing the wiki in KiWi, so the KQB has no need to include any browsing or exploration capabilities of its own.

And while result visualization through visKWQL could be considered for the future, it is not included in the prototype since KWQL itself is still early in its implementation phase as of writing this. Also, the results can be browsed like any other wiki pages, so that result visualization is not really necessary.

The only user interaction within the KQB is thus the query construction and editing, which is based on "Matching", since a visKWQL query describes the structure of the intended result (in the form of nested boxes) together with restrictions on them (in the form of values within keyword boxes).

One of the design goals for visKWQL and the KQB was user interaction in a consistent, easy to learn and understand way, based on the *Drag & Drop* approach that should be familiar to most users.

In the implementation of the KQB, care has been taken to make the query construction into a smooth experience, with all actions, apart from entering text into text fields, based on

[10] the rule would create a new content item with a comma-separated list of all authors of content items with "KiWi" in their title

drag & drop or click operations with the left mouse button, and not breaking the experience with the requirement to use context menus or other methods.

The available user actions based on drag & drop or simple left clicks with the mouse are:

Element Creation: to create a new element (either a single box representing a KWQL construct like a resource, a box with a child, like a qualifier box that already includes an input-box for user convenience, or a complete example query) the user can click on the element in a simple drop-down menu and it will appear on a free position within the workarea

Element Deletion: to delete an element, it can simply be dropped outside the indicated workarea

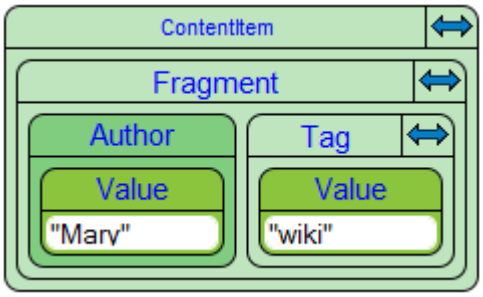
Child Addition: when a box is dropped on the body of another box (and if it is a valid child for this box) it will be added as a child element. When a child is added, all other boxes in the query are automatically resized to let it fit in

Child Removal: a child box can simply be dragged out of its parent box and dropped on the workarea (or on the body of another box, or it can be deleted) to remove it as a child. Similar to when a child is added, when it is removed, all other boxes in the query will be resized again

Type Switch: when a box is dropped on the label of another box, and the two boxes represent KWQL elements of the same type (for example an AND and an OR element, or two qualifiers with the same underlying data type, for example Text and Title, which are both strings) the two boxes switch their type. This way, it is possible, for example, when one has a complex query with an AND box in the middle, to turn that box into an OR just by dropping an OR box on its label, without the need to drag all the child boxes of the AND to the OR and replacing the AND in its parent box

Text Switch: similar to the type switch, when one Value box is dropped on the label of another, their text values will exchange. When a Variable box is dropped on the label of another Variable box, the variable names are switched

Information Hiding: one of the design goals was information hiding. This too has been realized in an easy to grasp way, by clicking once on the label of a box to hide its body (collapse it), and clicking again on the label to display it again. This way, boxes with an arbitrary number of descendants can be reduced to just one box label, making complex queries more comprehensible and preventing them from becoming too large for the screen. Since all boxes normally have a label and a body, it is also easy to see on a glance which boxes are currently collapsed. The system also saves the state of all child boxes, so that when a box's child box is collapsed and the user clicks twice on the parent box's label to hide and restore its body, afterwards the child box is still collapsed, and will only change when its own label is clicked. Figure 3.6 shows a simple query before (Fig. 3.6a) and after (Fig. 3.6b) clicking the label of the Fragment box to hide its body



(a) No hiding

(b) Fragment is collapsed

Figure 3.6: Information hiding in visKWQL

3.3.2.3 Roundtripping

One of the most important design goals of visKWQL was roundtripping, the possibility to switch between visual and textual query representation, to be able to edit the query in either of them, and see the changes reflected in the other.

Roundtripping has been fully implemented, allows the user to edit a query in both representations, and has the additional advantage of helping the user to learn the textual KWQL language by visually editing visKWQL queries, due to the side-by-side display of both representations.

The first part of roundtripping is the mapping from visKWQL to KWQL.

For this end, the system translates the visual query to the textual one after every user input action (like dropping an element somewhere or entering a character in a text field) and immediately displays it to the user, or shows an error message if the query is not currently syntactically correct.

For the other direction, the mapping from KWQL to visKWQL, the system includes a KWQL parser.

The user can, at any time, edit the current textual query, or enter a new one, and upon a button click the system will parse the query and replace the current visual query with it, or generate an error message if the textual query is not valid KWQL.

For more information about roundtripping, please see chapter 3.4.2.

3.3.2.4 User Guidance

As explained in chapter 3.3.1, supporting the user throughout the query construction process is one of the key aspects of visKWQL and the Query Builder.

A first aspect of user guidance is to never let the user feel “alone”. Especially when new to the system, the user should not be in a position where he does not know what to do. The KWQL Query Builder thus has a *Hint Pane*, a text area below the editor workspace, where helpful information is displayed to the user. It is mainly used to give hints about what to do when there is a problem in the query (see section 3.3.2.6), but it also displays a “welcome message” when the KQB is first loaded, which can be seen in Figure 3.7.

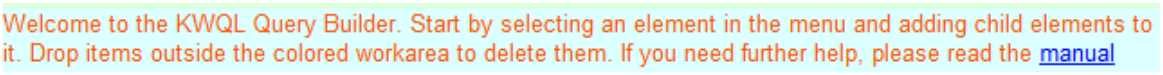


Figure 3.7: The KQB hint pane

The context-sensitive help provided in the hint pane is one step of making the system self-explaining.

The second step towards that goal are tooltips, which give the user information about elements on the screen.

Tooltips come in two variants. The first variant are explanations of the KWQL elements within a query. These tooltips are always displayed when the user hovers the mouse over a box, in a special text area, the *Tooltip Pane*, which is located at the bottom of the editor workspace, making these tooltips relatively unobtrusive, and only displaying text when the mouse hovers over some box.

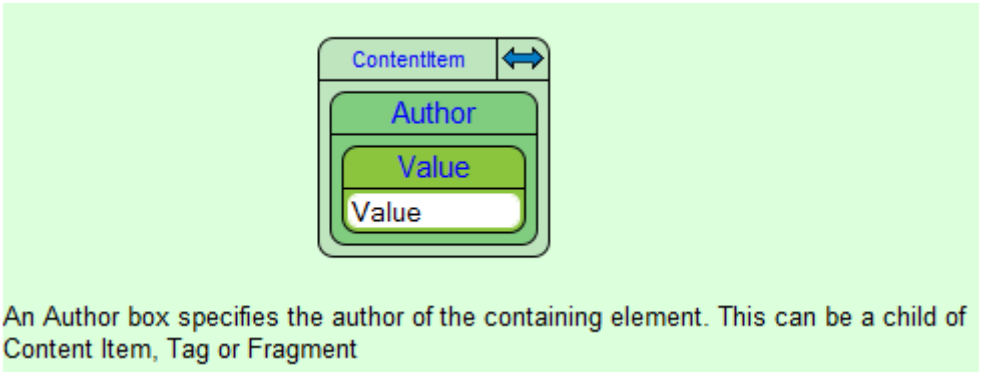


Figure 3.8: The KQB tooltip pane

Figure 3.8 shows the tooltip displayed to the user when he moves the mouse pointer over the Author box in the query.

The second type of tooltips are those which don't give general explanations about a box, but which give information about its current state.

These tooltips are displayed in an obvious way, directly next to the box the tooltip is associated with.

They give information about the validity or invalidity of dropping actions while a box is being dragged (see section 3.3.2.5), or display warning or error messages when a problem was detected in a box the mouse currently hovers over (see section 3.3.2.6).

An example for such a tooltip is given in Figure 3.9. The KQB workarea, in which the user edits his queries, is a large green rectangle, taking up most of the screen[11]. The white area, where the Value node in the example is currently being dragged, is "outside" the workarea, where boxes can be dropped to be deleted, which is indicated to the user through the tooltip.

[11] please note that its green background has been omitted in the other pictures in this chapter for higher image clarity

Figure 3.9: A KQB tooltip

For user convenience, the complete text of a textbox is also displayed in a tooltip when it exceeds the length of its containing textbox.

Finally, when the user selects boxes to be added to the workspace from a drop-down menu, the informational KWQL tooltips are also displayed next to the mouse pointer when it hovers it over a menu entry, so that the user can read an explanation of an element before he actually adds it to the workspace.

3.3.2.5 Error Prevention

Another important aspect of supporting the user during query construction is the prevention of (syntactical) errors.

The KQB uses two devices to inform the user about the validity of actions and the state of query elements: colors and tooltips.

Three colors are used to indicate different states:

Green signals a valid action

Red indicates an error

Orange stands for "warning": there is a problem, but it is not as severe as an error

These colors were chosen since they should be understandable in their meaning by people from most cultures, since for many they are part of everyday life (e.g. in traffic lights). As already shown in the last section, in addition to colors, the KQB also displays tooltips in situations where they are helpful, to present the user with further information[12].

To prevent syntactical errors, the Query Builder only allows the user to create valid box nestings, and prevents him from adding child boxes where they are not allowed, and from switching types between nodes of unequal type.

When adding a child to some box, there are three possible alternatives, all of which are indicated through color and tooltip to the user:

[12] and color indicators are never used without accompanying textual tooltips, so that even color-blind people should never be in the unclear

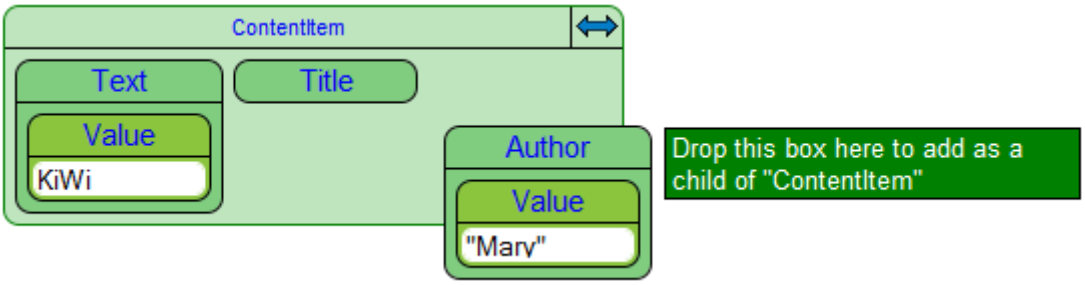


Figure 3.10: Valid drop location

Figure 3.10 shows the first possibility: the Author box is dragged over free space within a ContentItem. Author is a valid child of ContentItem, and can thus be dropped here. This is indicated by a green border of the ContentItem, and a green tooltip.

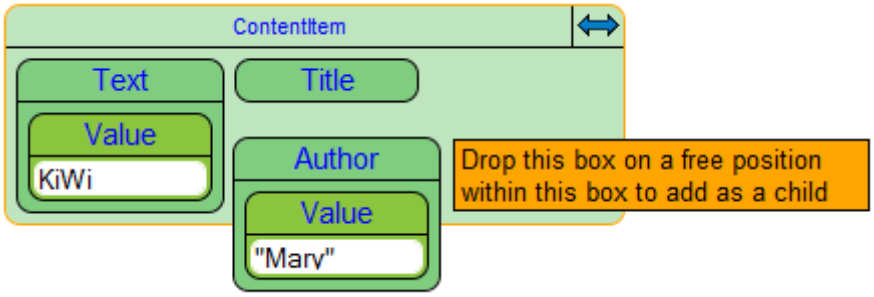


Figure 3.11: Invalid drop location within a valid parent box

Figure 3.12 shows the second possibility: the Author box is dragged over the ContentItem box, which is a valid parent. But the Author box is currently in an invalid position within the ContentItem, occupied by a Title child. This problem is displayed to the user through an orange border around the parent box, and an orange tooltip. The decision for this behavior, instead of just letting the user drop a child in this position, has been made because in this case it would be unclear if Author should become the second or third child of ContentItem. Within ContentItem, this would make no practical difference, but in some boxes, the order of the child boxes is important. With the chosen approach, the user sees that he can not drop Author in this position, and can either drop it to the right of the Title child, or he can first move the Title child to the right, and then drop Author between Text and Title.

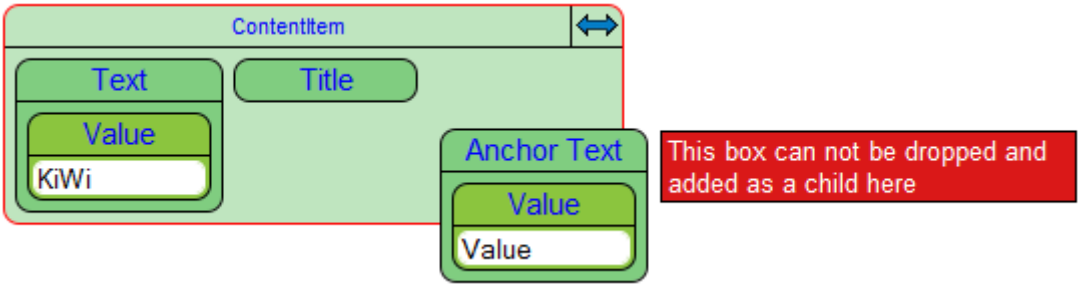


Figure 3.12: Invalid drop location

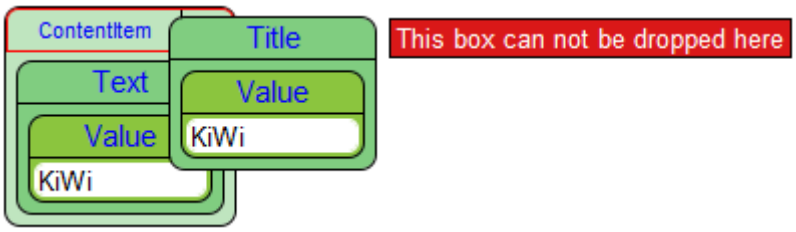
The last alternative for a drop, pictured in Figure 3.12, is a drop that would lead to an invalid nesting. In the example, the user has dragged a Link's AnchorText qualifier over a ContentItem, where this qualifier is not allowed. This is displayed to the user with a red border around the parent box, and a red tooltip indicating the problem. If the user drops the AnchorText box in this position, the system will not allow it to be added as a child of ContentItem, but return the box to the position where the user picked it up.

The system checks for the validity of a child addition on all nesting levels, and not only the direct parent. So, even if a content item contains a nesting of, for example, AND and OR boxes, the system does not allow dropping the AnchorText on one of these nodes, as long as ContentItem is in the parent hierarchy. Similarly, when the user first drops an AnchorText on an AND node, the KQB will not allow him to drop this AND node on a ContentItem node while it contains a child that is not valid for ContentItem.

The system checks the validity of drop actions on every mouse movement while a box is being dragged, so that when dragging the Author box over the ContentItem box in the example, both border color and the tooltip (which itself moves together with its associated box, when the box is dragged) will immediately change between orange and green when the user moves the mouse between occupied and free space within the parent box.

In addition to child additions, type switches could also lead to syntactical errors in the form of invalid nestings, and their validity is therefore checked by the system when the user drags a box over the label of another box. Similar to when dragging a box over another box's body, the system informs the user through colored border and tooltip if a drop action is allowed in the current situation. In this case, however, the colored border is put only around the label and not the whole box, to clearly indicate to the user that he is dragging over the label and not the body of a box.

Figure 3.13 shows a situation in which dropping is not allowed (Fig. 3.13a), since Title and ContentItem are qualifier and resource, and can not switch type, and a situation in which dropping is allowed (Fig. 3.13b), since Text and Title are both string-valued qualifiers and can be switched.



(a) Dropping not allowed

(b) Dropping allowed

Figure 3.13: Dragging over a box label

3.3.2.6 Error Reporting and Correction

Not all syntactical errors within a query are preventable. For example, a qualifier with the value `"KiWi` would include an error, since the string contains only an opening quotation mark, but no closing one. It is impossible to not allow the user to type in this value, since he might still add the closing quotation mark. Another problematic case would be a rule which uses a variable in its construct part that was not defined in the query part. This would constitute an error, but to prevent the user from making it, he would always be required to first form the query part of a rule and then the construct part. Enforcing this and similar things would lead to less flexibility, and would stand against the design goal of high user friendliness.

Thus, the KWQL Query Builder allows the user to make actions that lead to errors of this sort. But, it still detects them, and visually informs the user about problems in his query.

In detail, the possible errors which the Query Builder detects are:

- Invalid variable names or keywords (e.g. containing invalid characters like spaces in a variable name, or missing quotation marks in keyword strings)
- Empty strings
- Invalid value format (e.g. a letter in the SOME operator child that represents the numerical value associated with the SOME)
- Misplaced operators (operators like ALL or SOME may only occur in the construct part of a rule, OPTIONAL only in the query part)
- Undefined variables (variables used in the construct part of a rule without being defined in its query part)
- Invalid number of children (for elements like the SOME operator which need to have a specific number of child elements)
- Invalid child (during visual query construction the system prevents actions that would lead to this error, however when parsing textual KWQL code, the decision has been made to allow parsing invalid resource:qualifier pairings and then visually displaying the problem and letting the user solve it, rather than just generating an error message and requiring the user to correct the textual query first)

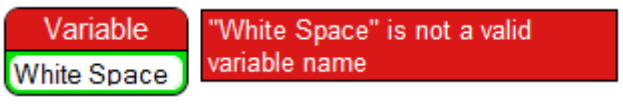


Figure 3.14: A box containing an error

When a query contains an error, this problem is immediately visualized to the user through multiple means:

- Instead of the textual KWQL query, the user is displayed the error message of the first error in the query
- All boxes containing an error have their labels colored in bright red

- When the mouse is moved over a box containing an error, the error message associated with that box is displayed in a red tooltip (see Figure 3.14)
- The hint box below the workspace displays a more detailed explanation of the first error in the query, and informs the user how to correct it (in the example, the hint pane would display a message that white spaces are not allowed within variable names)

To help the user find an error, especially when parts of his query are hidden, the KQB also colors the labels of all parents of a box that contains an error in red, and displays a tooltip informing the user that a child box contains an error, when the mouse is moved over one of its parent boxes (see Figure 3.15).

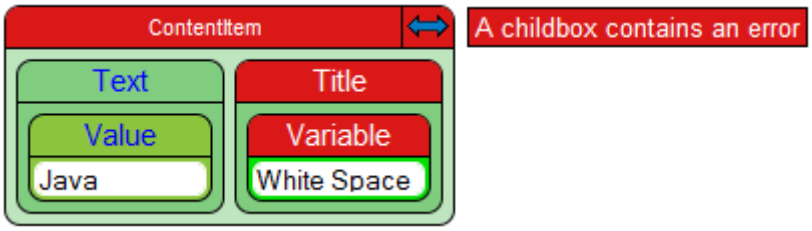


Figure 3.15: A childbox contains an error

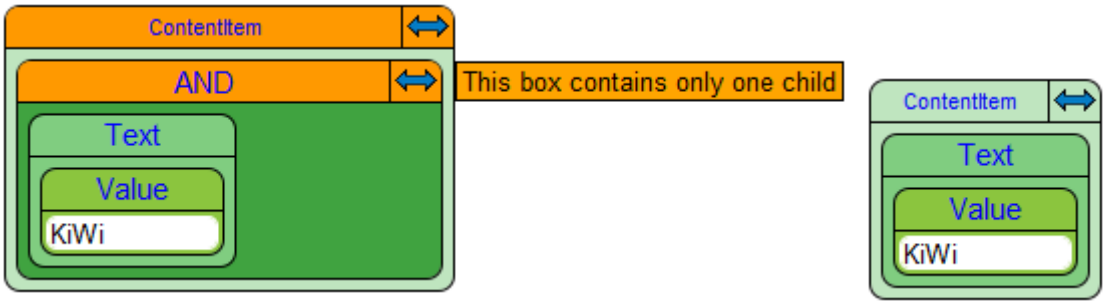
In addition to the above errors, visKWQL queries can contain another category of less severe errors, which the Query Builder automatically corrects.

These category includes:

- AND or OR boxes with only one child
- Rules with only one child
- Empty boxes
- Unplaced variables or operators (variables and some operators, like ALL, can only occur within a rule, and are thus considered “unplaced” while they are not child elements of a rule)

The Query Builder finds and corrects these problems for the user, and displays valid textual KWQL output, even when a query contains one of the above problems.

The system internally removes the problematic boxes while it translates the visual query to KWQL. Empty or unplaced boxes are completely removed, while in the case of AND or OR boxes with only one child, the box itself is removed, but its child is appended to its parent box.



(a) Query containing a problem (b) Corrected query

Figure 3.16: Problem correction in visKWQL

Figure 3.16 shows an example of a query containing an AND box with only one child (Fig. 3.16a) and how the system corrects it (Fig. 3.16b). Note that the corrected state exists only internally and is used for the translation to the textual KWQL query, since the checking is done upon every user action, and the system mustn't remove boxes while the user is still working on the query. However, thanks to roundtripping, the user only needs to press the "Parse Query" button of the KQB, to parse the textual KWQL generated by query 3.16a, to have the KQB replace the problematic query with its corrected visual form of Figure 3.16b.

Similar to the colors used during drag & drop, while errors are reported through bright red labels and tooltips, the less severe problems, which do not require the user to change his visual query to get a correct textual query, are represented through an orange coloring.

When such a problem is detected, the hint pane will, similar to when an error is detected, display an explanation of the problem and how to correct it, and also inform the user that the offending box is being ignored for the generation of the textual KWQL query, giving him the chance to learn making entirely correct queries, but not absolutely requiring him to do so for problems which can be corrected by the system.

3.3.2.7 Other Features

Variable Highlighting

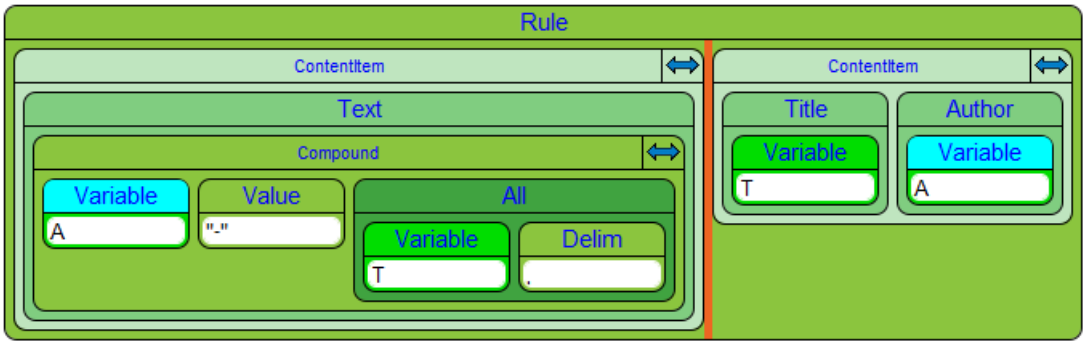


Figure 3.17: Variable Highlighting

One feature that has been implemented to help the user when editing rules with multiple variables is *Variable Highlighting*.

When the mouse is moved over a variable box, the KQB checks if there are additional occurrences of a variable with the same name within the same rule. When such a variable is found, all of its occurrences are colored in a bright cyan, making it easy for the user to find equal variables. See Figure 3.17 for a query in which the user currently points the mouse over one of the "A" variable boxes.

In cases where part of the query is hidden, the user isn't required to unhide everything to find all occurrences of a variable, but the system colors the lowest parent(s) of fitting variable boxes in cyan, to indicate where the variable occurrences can be found. In Figure 3.18, the Text box has its content hidden, but contains a second occurrence of the variable "T", so when the user hovers the mouse over the "T" variable box, the Text box is highlighted.

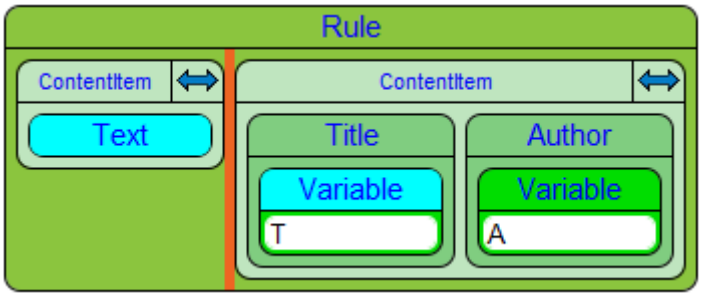


Figure 3.18: Variable Highlighting with hidden content

Resizing the Workspace

When the browser window in which the KQB is running is resized by the user, the Query Builder automatically resizes, to fit completely into the window. But since a user might prefer to have a larger editor workspace and scroll through it, the KQB allows the user to manually resize the workspace. This is done through drag and drop of the corner of the workspace, similar to resizing windows on the desktop of the operating system.

Saving Queries

Users might want to save and load queries which they might use multiple times, or save a query to work on it again at a later time.

To support this, the KWQL Query Builder allows the saving and loading of the currently displayed query. To do this, the KQB is integrated with KiWi, so that users who are currently logged into the KiWi system are displayed a list of their saved queries when they start the KQB, and can load them at any time, delete saved queries, and save their current query to the system for later use.

Undo & Redo

Undo and Redo functionality is a feature commonly found in editors, to allow the user to undo changes and return to a previous version of his document.

The KWQL Query Builder fully supports Undo and Redo for an arbitrary number of operations (there is no limit on how many operations can be undone or redone). To achieve this, the KQB saves the state of the visual query after every user action: dragging and dropping, key presses, clearing the workspace, loading a saved query, and parsing a textual query.

Each of these actions can thus be undone or redone through button clicks.

Browser Compatibility

One design goal of the system was to make it usable by everyone who could use KiWi, from within the user's web browser, without the requirement to install additional software.

To achieve this end, visKWQL and the KQB have been fully implemented in DHTML and run completely client-side, with the only exceptions being the saving and loading of queries, and query execution, for which the KQB communicates with the KiWi server.

Since the KQB is graphically intensive and uses modern Java Script methods, it might not run problem-free on browsers that are more than a few years old, but it has been tested with the most common currently available browsers and works without problems on them.

It has been tested and is guaranteed to work on

- Internet Explorer 8
- Google Chrome 2.0
- Mozilla FireFox 3.0
- Opera 10.0
- Safari 4

and newer versions. But care has been taken to implement the KQB in a cross-browser friendly way, so it should also work on some earlier versions of the named browsers, and on less common ones it was not tested on.

3.3.3 Query Builder Usage

3.3.3.1 The KQB User Interface

The KWQL Query Builder user interface includes the complete functionality to edit visKWQL queries and provide help to the user while he is working with the system.

The user interface consists of the following parts:

1) The Menu Bar The Menu Bar allows access to all the different visKWQL constructs, which can be used to form a query. They are grouped into Resources, Qualifiers, Operators, Other items, and Examples (please see section 3.3.3.2 for an explanation of those constructs). The menu is a simple drop-down menu, which displays items or sub-menus when the mouse is hovered over it. The Qualifiers are grouped after the Resources they can belong to (see Fig. 3.20) to help select appropriate Qualifiers for the right Resources. When a menu item is clicked, the appropriate visKWQL box is created and added to the workspace

2) Undo & Redo These buttons allow to undo or redo the last action. There is no limit on how many steps can be undone or redone, and every action, like dragging and dropping a box, or entering a character into a text field, is considered one step

3) The Workarea This is the main workspace, on which visual queries are displayed and edited

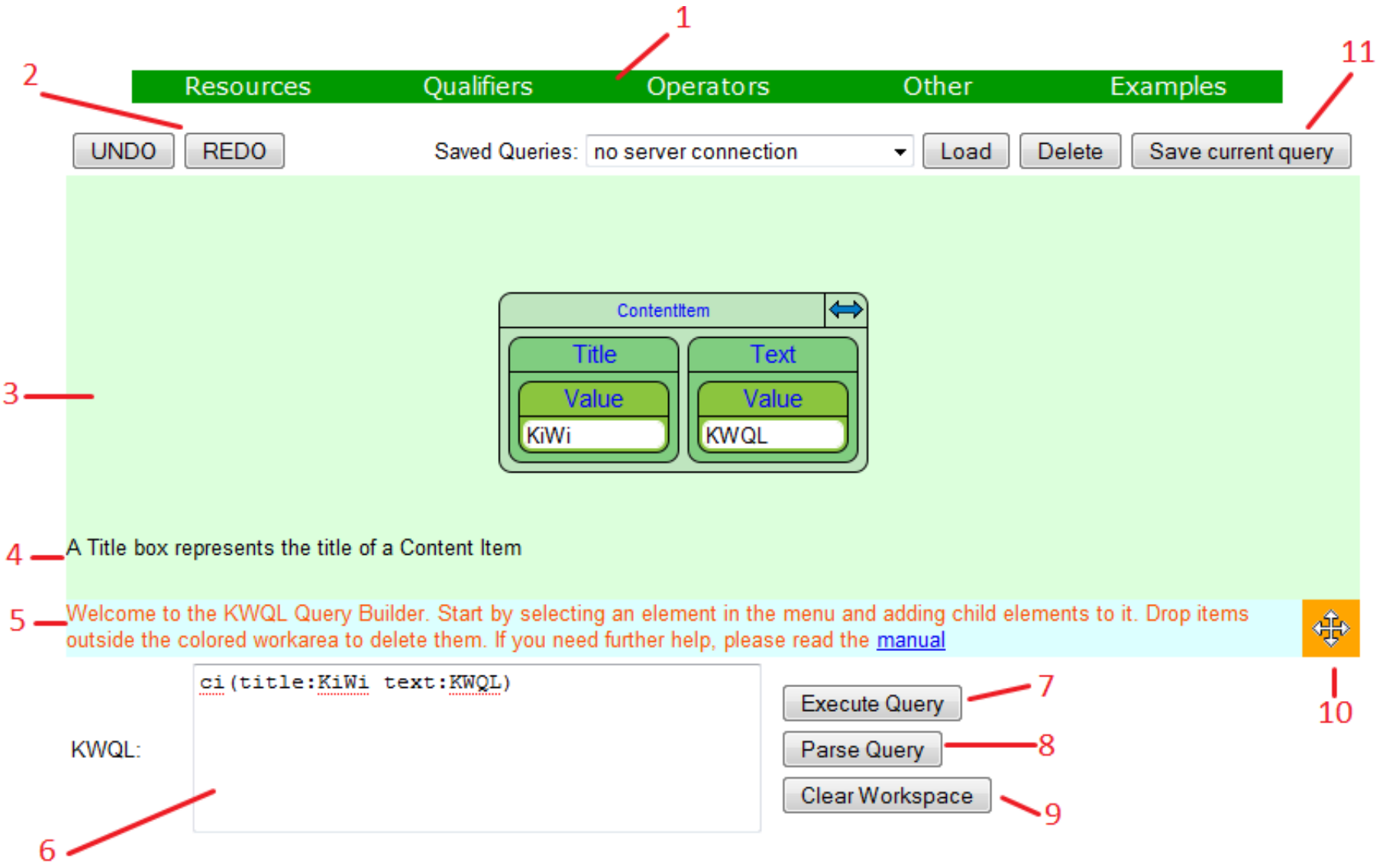


Figure 3.19: KWQL Query Builder GUI

4) The Tooltip Pane When the mouse is moved over a visKWQL element in the workarea, the Tooltip Pane displays a description of that element

5) The Hint Pane The Hint Pane provides help in the form of hints of what to do next, or error explanations and suggestions of how to solve an error or problem that might exist in the currently displayed query

6) KWQL Output In this field, the textual KWQL query is displayed. It is updated after every user action. If the current visual query is syntactically incorrect, an error message will be displayed in its stead

7) Execute Query Button Pressing this button will execute the current query (assuming it does not contain an error), and open the result page

8) Parse Query Button This button will parse the textual query currently in the KWQL Output text field (which the user can enter or edit) and replace the current visual query with it

9) Clear Workspace Button The Clear Workspace Button simply deletes both the visual and textual query. This action can be undone with the Undo Button

10) The Resize Box This box can be dragged with the mouse to change the size of the Workarea

11) Saved Queries When logged into KiWi, the drop down box displays a list of saved queries. When a list entry is selected, and the Load button is pressed, that query will be loaded and replace the current query in the Workarea. The Delete button will remove the current list entry and delete the saved query. The Save button will ask for a name, and then save the currently displayed visual query under that name on the server

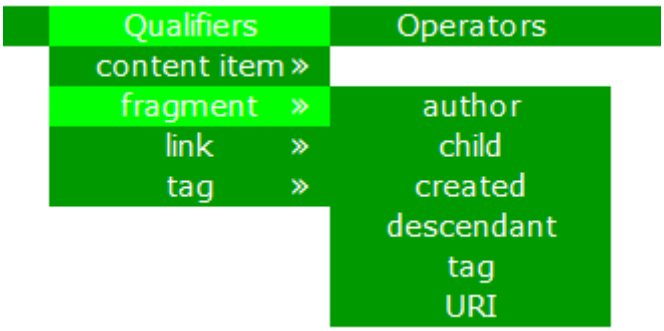


Figure 3.20: A part of the KQB menu

3.3.3.2 (vis)KWQL elements

Every visual element of visKWQL corresponds to one textual element of KWQL. To be able to create meaningful visKWQL queries, it is thus important to understand the meaning of those elements. While the KQB tooltips give a short explanation of all elements, so that the user is not required to read a manual to be able to construct useful queries, a better

Resource	content item	fragment	link	tag
Qualifiers	URI	URI	target	URI
	author	author	anchorText	author
	created	created		created
	lastEdited	descendant		name
	title	child		
	text			
	numberEdits			
	descendant			
	child			
Subresources	fragment			
	link	link		
	tag	tag	tag	

Figure 3.21: Resources and accompanying Qualifiers and Subresources (from [69])

understanding of the underlying conceptual model and the meaning of different language elements will be helpful, so that a full listing of all elements is given here.

Resources

In KiWi's conceptual model, the wiki structure is made up of four different resources. These resources are directly queryable by KWQL and visKWQL:

Content Item (ci): KiWi's most basic and most important structure is the *Content Item*[13]. Content Items are holding textual or multimedia data, and correspond to wiki pages. While each Content Item can be seen as one wiki page, a Content Item can also include other Content Items, which in turn can include other Content Items, so that each Content Item can be a tree of Content Items. An example would be a wiki page for "Elephant" with a textual description and a photo of an elephant. While the whole wiki page would be one unique Content Item, this Content Item would include one Content Item with the textual description of the elephant, and one Content Item with the photo. When the photo is clicked in the "Elephant" wiki page, the elephant photo's own wiki page (Content Item) will be displayed. One Content Item can thus include a series of other Content Items, however, there are neither overlap nor cycles in Content Item compositions

fragment: A *Fragment* is a piece of textual data within a Content Item. Its length is arbitrary, it can be one word, one sentence, one paragraph, or any other meaningful portion of text that a user might want to annotate. Fragments can include other Fragments, but there is no overlap, and one Fragment can not include text from more than one Content Item

[13]the lower case or abbreviated name, like ci, of an element as given first in the list is always how the element is named in KWQL

Link: *Links* in KiWi are the same as hypertext links on web pages. They have a target Content Item and an anchor. When a user clicks the anchor on a wiki page, he will be directed to the target wiki page

tag: *Tags* are the annotations in KiWi. Users can annotate Content Items, Fragments and Links. Each Tag has a name, which is free-form information or meta-data which the user gives to some resource. For example, an "Elephant" Content Item could be annotated with the Tags "animal", "lives in Africa", "grey" or any other fitting information

Qualifiers

Each Resource has accompanying data and meta-data, some of which is automatically created by the system (like the creation date of a Content Item), and some of which is created by users (like the text of a Content Item). These are called Qualifiers of the Resources, and are also directly queryable by (vis)KWQL:

author: Content Items, Fragments and Tags each have an Author qualifier. This Qualifier is a list of the names of the users who have authored that particular Resource

created: The Created Qualifier is the date that a Content Item, Fragment or Tag was first created

lastEdited: This Qualifier specifies the date a Content Item was last edited

tag: The list of Tags associated with a Content Item, Link or Fragment

title: The title of a Content Item

text: The text of a Content Item

numberEdits: The number of times that a Content Item has been edited

link: A Link within a Content Item

fragment: A Fragment within a Content Item

child: A Content Item or Fragment that is a direct child of another Content Item or Fragment, meaning it is directly included in it

descendant: A Content Item or Fragment that is a direct or indirect child of another Content Item or Fragment, either directly included in it, or included within another child

URI: A Unique Resource Identifier is associated with every Content Item, Fragment and Tag. Through an URI, each of these Resources is uniquely identifiable and reachable

disagree: The Disagree Qualifier is a list of user names who do not agree with the content of a Content Item or with a Tag

target: The Content Item a Link points to

anchorText: A Fragment within a Content Item, to which a Link is anchored

name: The Name Qualifier specifies the text of a Tag

Operators

Operators in (vis)KWQL are associated with one or more other language elements and allow the creation of complex queries:

AND: The AND operator forms the logical conjunction of two or more other elements. In visKWQL, all children of an AND box must be true in a query. For example, a query in which author:Mary and author:Carl are both children of an AND within a Content Item, will only return Content Items which have been written by both Mary and Carl together, but will not return ones where only one of the two was the author (visual examples for the use of operators will be given in the next section)

OR: The OR operator forms the logical disjunction, where only one of its children must be true, but not all of them

NOT: The logical negation. For example, if a visKWQL query includes author:Mary as the child of the NOT operator within a Content Item, the query will return all Content Items where Mary was not the author

COUNT: This operator can only have a variable as its child, and will return the number of values within that variable. This operator is only meaningful in the construct part of a Rule

ALL: The ALL operator will return (in the construct part of a rule) all values saved in a variable. It has two children: the variable whose values it extracts, and a string which is used to delimit those values. For example, if an ALL operator has a variable child containing the values x, y and z, and a delimiter child " - ", it will return the string "x - y - z"

SOME: The SOME operator works like the ALL operator, only that it has an additional numerical child, which sets the maximum number of values to return. The COUNT, ALL and SOME operators can all be used to construct textual values, like a Content Item's Text or Title, within a rule

OPTIONAL: The OPTIONAL operator allows the specification of an optional element within the query part of a rule. It has two children: The first child is a default value. The second child is a variable assignment to some Qualifier. When the Qualifier is present and has a non-empty value, this value will be bound to the variable. If it is not present or has an empty value, the default value will be bound to the variable

Other elements

Some additional elements of (vis)KWQL do not fall into the above categories. These are:

Value: A simple string, which is either the value of some Qualifier or Operator, or a Keyword on its own. As a value of a Qualifier within a query, it can be seen as a value restriction. A query consisting only of the Qualifier "author" with the value "Mary" will only return those Content Items which have an author with the name Mary

Rule: A rule allows the creation of new Content Items. It consists of two parts: a query and a construct part. The first child (in visKWQL) is the construct part, and specifies a Content Item that is to be created. The second child is a normal query, which usually contains variable definitions which are used in the construct part (please see the next section for an explained visual example)

Variable: A variable definition to be used in a rule. In the query part, a Qualifier can have a variable definition as a child, instead of a value. In this case, the Qualifier's value is bound to this variable (or the Qualifiers' values are bound to it, if the query results in multiple items). In the construct part of the rule, these values can than be extracted with the ALL or SOME operators

Binding: A (variable) Binding is a combination of value restriction and variable binding. In visKWQL, a Binding contains two children. The first is a value, which restricts the Qualifier the Binding is associated with. The second child is a Variable, to which the actual value will be bound. This way it is possible, for example, to bind all Content Item Titles which include the word "wiki" to a variable and construct a new Content Item with links to all those wiki-themed Content Items, with the link anchor being the full title of the specific Content Item

SPARQL: KWQL allows the inclusion of SPARQL queries. In visKWQL, this can be achieved by entering the textual SPARQL query into a SPARQL box

Compound: The Compound is an element specific to visKWQL. In KWQL, a Qualifier in a query can only have one child (a Value, Variable, or Binding), but in a rule's construct part, some Qualifiers like Text and Title can have multiple children. Qualifiers in visKWQL only have one child, so to allow them to have multiple children within a rule's construct part, multiple elements can be put into a Compound box, which can then be added as a child of the Qualifier

3.3.4 Query Construction - User's Manual

3.3.4.1 Creating Queries with Drag & Drop

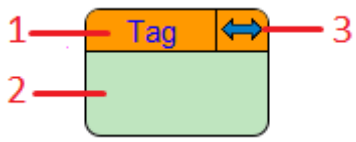


Figure 3.22: A visKWQL box

All queries in visKWQL are created by nesting boxes by dragging and dropping them onto each other.

Boxes represent all language elements, and work in the same way, no matter if they represent a Resource, an Operator, or something else.

Figure 3.22 shows an empty box. All boxes have a *Label* (1) and a *Body* (2). Some boxes additionally have a *Resize Box* (3).

The box label always shows the name of the box, which is the KWQL element (Resource, Qualifier, Operator, ...) it represents. The box body can hold child elements, which are either other boxes or text boxes, in which keywords, values, or variable names can be entered.

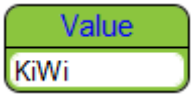


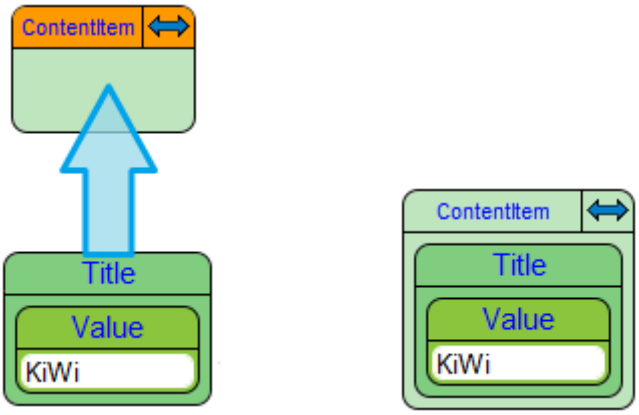
Figure 3.23: A keyword query

The simplest kind of query in visKWQL is a keyword query, which results in a search for the keyword over the entire wiki.

For a keyword query, select "Value" in the "Other" category of the menu bar. Clicking it will create a new Value box in the workarea. Now enter a keyword into its text field, and the query is finished. Figure 3.23 shows a keyword query for "KiWi". When executed, this query will result in a list of all Content Items (which are the default resource if the query doesn't explicitly declare one) which contain the word "KiWi" in any Qualifier.

The real power of (vis)KWQL, however, lies in its capability to directly query the structure of KiWi, specific Resources and Qualifiers.

To create a more specific query in visKWQL, a box can be added as a child box to another box, by dragging and dropping one box on another box's body (see Figure 3.24).



(a) Dragging into the body of another box

(b) The result

Figure 3.24: Adding a childbox

Assume that we want to find all wiki pages which have the word "KiWi" in their title.

First select "ContentItem" (since a Content Item represents a wiki page) in the Resource menu, and "Title" in the Qualifier menu, to add them to the workarea. Note that for convenience, Qualifiers like Title already include a keyword (Value) child when created.

Next, enter the keyword we are searching for, “KiWi”, into the text box of the Title.

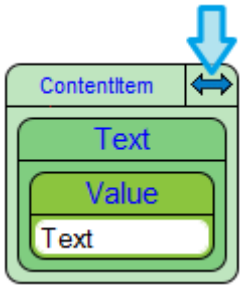
Now add the Title box as a child to the ContentItem box. To do this, “grab” the Title box by clicking the mouse down on its label, and drag it over the body of the ContentItem box (see Figure 3.24a). Let the mouse button go to drop it there. Figure 3.25c shows the result of the action. If the query is executed, it will return all wiki pages with the word “KiWi” in their title. You can also search for wiki pages where the title does not include “KiWi”, but where the title is exactly “KiWi”, by putting the keyword into quotation marks.

Note that, since Content Item is the default Resource, the Title box alone would produce the same query results as the Title box within a Content Item box.

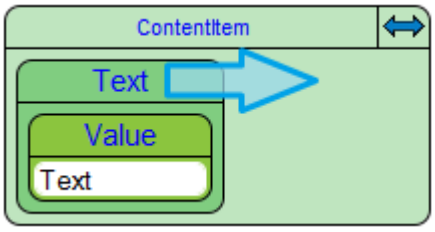
When you create queries this way, the system will check if the result of a drag & drop operation is valid, and will prevent you from dropping a box in places where that would result in an invalid query.

Also note that, when there are multiple (not nested) boxes on the workspace, only the box that was put first on the workarea is considered the current query and will be shown in its textual form in the KWQL Output field, and executed when the Execute Query Button is pressed.

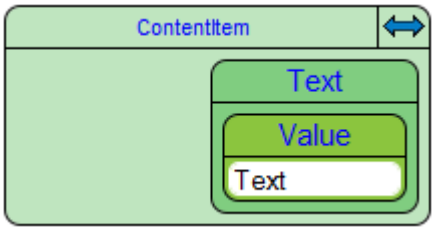
To remove boxes from the workarea, simply drop them outside the workarea to delete them. You can also delete everything by pressing the “Clear Workspace” Button. If you accidentally delete something, you can use the Undo Button to restore it.



(a) Clicking the Resize Box



(b) Dragging inside the box



(c) The result

Figure 3.25: Expanding and dragging within a box

Some boxes, like those representing Content Items, have a Resize Box in their label, indicated by a two-headed arrow. Boxes with such a Resize Box can have multiple children.

To add a second child to such a box, click the Resize Button (Figure 3.25a) to expand it. Now the box is wider, and has a free place in which another child box can be dragged.

It is also possible to arrange the children of a box in a new order, by dragging them within their parent box. As the example shows, in an expanded box, you can simply drag one child to a free position to place it there (Figures 3.25b and 3.25c). Now a second child can be dropped in the place the Text box has occupied before. You can click the resize box again at any time to hide the free space.

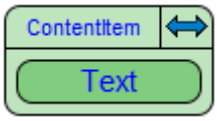
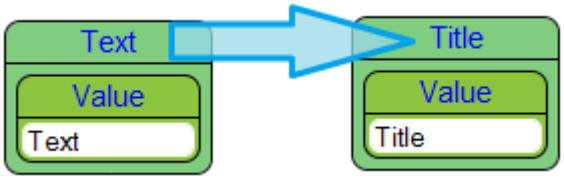


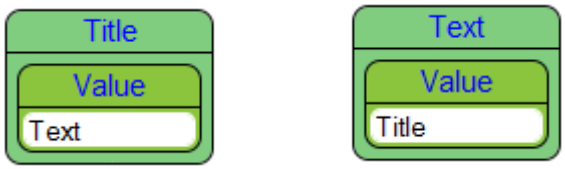
Figure 3.26: Hiding the body of a box

If you create more complex queries, they might get quite large and unclear. To hide parts of a query, you can click the label of any box to fold it together, to hide its body and all child boxes. Figure 3.26 shows the result of clicking on the label of the Text box of the query in Figure 3.25a. Clicking it again will show the body and child boxes again. It is possible to hide arbitrarily large parts of the query this way, making working with large queries easier.

The KWQL Query Builder also allows dropping a box on the label of another box, to switch the two boxes.



(a) Dragging and dropping on the label



(b) The result

Figure 3.27: Switching boxes

Figure 3.27 demonstrates this. Assume we want to search the Text of a Content Item instead of its Title for a specific keyword. One way to do this would be to drag the Title box out of the Content Item, then drag and drop the Text box on its body, then enter the keyword in the Text box.

A faster way to do this is to switch the Title and Text boxes. To do this, simply drag the Text box, and drop it on the label of the Title box (Figures 3.27a and 3.27b). Note that not the complete boxes change place, but that they switch their type. The child boxes stay in place. The example shows the type switch between two free-standing boxes, but the switch is possible no matter on what level of a query the two boxes are, or how many children they have.

If you drag a box that contains a text input field, like Value or Variable, onto the label of another such box, their text values are switched.

A requirement for a successful switch is that the two boxes are compatible. Qualifiers with textual values, like Title and Text in the example, can be switched, as can an AND with an OR operator. Switching for example a Qualifier with an Operator box, however, would result in an invalid query.

The system thus checks the validity of all drop actions, and will only accept meaningful switches and prevent forbidden ones.

Problems that the system can not directly prevent will be shown to you through red or orange colored box labels. In such a case, move your mouse over the colored label to read the error message, and pay attention to the Hint Pane, which will inform you about the cause of the error and how to correct it.

3.3.4.2 Advanced Examples

In addition to the Resources and Qualifiers shown in the examples of the previous section, visKWQL also supports all KWQL operators.

The first kind of operators are connectives (the AND and OR operators) and negation (the NOT operator).

Assume you want to search for all wiki pages which do not include “KiWi” in their title. To create a query for this, put a Title box inside a NOT box, and then that NOT box inside a ContentItem box, instead of putting the Title box inside the Content Item box, which would be a query to return all wiki pages that do contain “KiWi” in the title.

If you want to search for all wiki pages which were either written by Mary or by Carl, put two Author boxes, with the values “Mary” and “Carl”, inside an OR box. Note, that in boxes that can hold multiple children, like a Content Item, an implicit AND is assumed, so that putting the two Author boxes inside an AND box within a Content Item would be equal to putting them both directly in the Content Item, to search for all wiki pages written by Mary and Carl together.

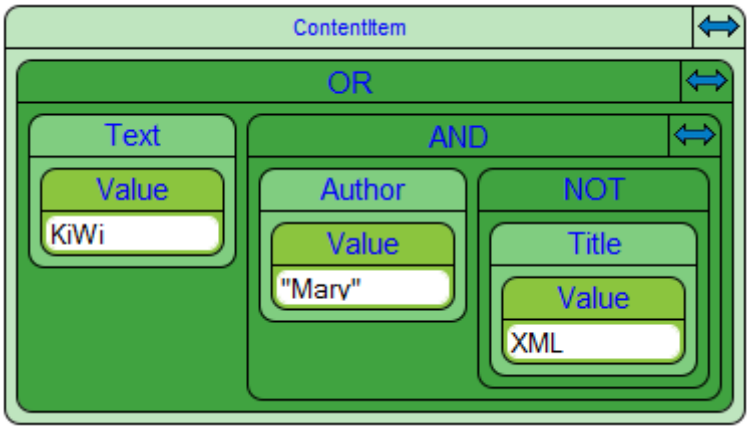


Figure 3.28: Using operators

Figure 3.28 shows a query that makes use of the AND, OR and NOT operators, which can be arbitrarily nested to produce the desired query. In the example, the resulting query returns all wiki pages with contain “KiWi” in their text, or which were written by Mary and do not contain “XML” in the title.

The other operators of visKWQL, ALL, SOME and OPTIONAL, can only be used within rules. Rules are used to create new content items. They consist of a query part, and a construct part which creates new data, based on the results of the query part.

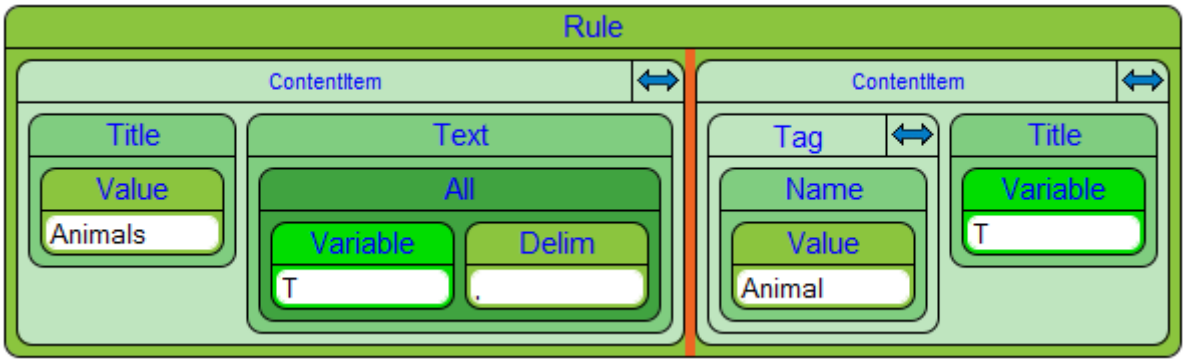


Figure 3.29: A visKWQL rule

Figure 3.29 shows a complete rule, which creates a new wiki page with a listing of all animals for which a wiki page exists.

The right side of the rule, the query part, contains a query as used before, only that it now contains a variable definition. In this example, the query will search through all wiki pages, and for those pages that have an "Animal" tag, it will save the title in the variable[14] T.

The left part of the Rule is the construct part. It creates a new wiki page with the title "Animals", and then uses the ALL operator to construct its text. The ALL operator extracts all bindings of its variable child. Its first child is the variable, the second child a textual string that is used to delimit the values of the variable. The resulting text in the example will thus be a comma separated list of all the titles of wiki pages which were tagged "Animal".

The SOME operator works very similar, only that it contains an additional child in which a number can be entered, and it will not return all but only that number of values.

Instead of the Variable box, you can also use a Binding box to restrict the variable binding further. A Binding box contains a Variable child and a Value child, and will only bind values to the variable if they fit with the Value child. For example, if a Binding with the value "grey" would be used in the example instead of the Variable, the resulting wiki page would only list the title of those pages with a tag "Animal" where the title also contains the word "grey", for example "Grey Mouse", or "Grey Wolf".

[14] as mentioned before, when selected in the menu, Qualifiers come with a Value child. To put a Variable box in its place, select a Variable from the "Other" menu, and switch it with the Value child

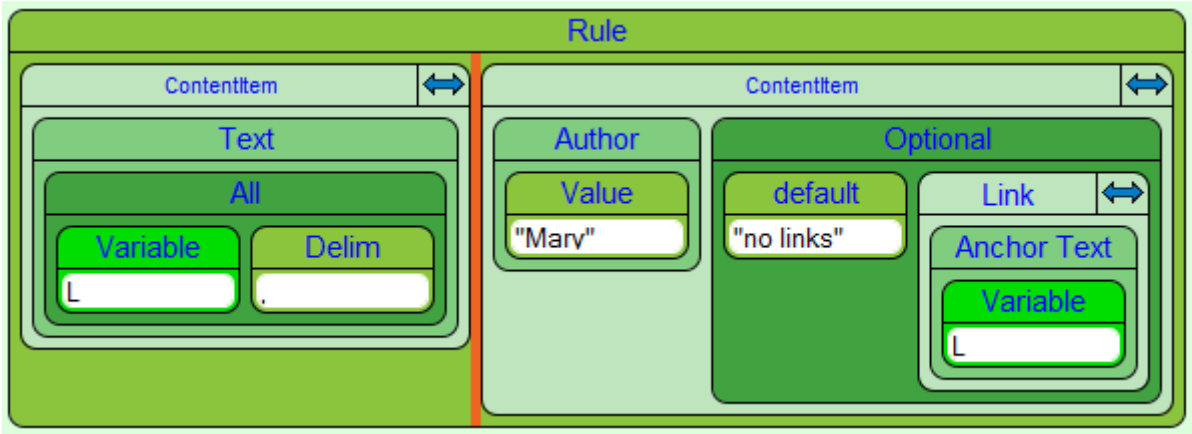


Figure 3.30: A visKWQL rule with the OPTIONAL operator

The last visKWQL operator is the OPTIONAL, which can only occur in the query part of a rule.

Figure 3.30 shows a rule that uses it. The first child of the OPTIONAL operator is a default value, the second child is the optional part of the query. The query searches all wiki pages written by Mary. If a found page contains a link, the link's anchor text is assigned to the variable L. If it contains no link, the string "no links" is assigned to L. The construct part simply creates a new wiki page with a comma-separated list of the anchor texts of links in Mary's wiki pages, or the text "no links" if there are no links on her pages.

3.4 Implementation

VisKWQL and the KWQL Query Builder have been implemented in DHTML (Dynamic Hyper Text Markup Language), using HTML for the KQB website structure, Java Script for the dynamic and interactive parts, and CSS (Cascading Style Sheets) to describe the graphical presentation. This way, the system can run completely client-side, within the user's web browser.

Only a small part of the KQB needs to interact with the KiWi server and is exempt from this:

The code that loads and saves visKWQL queries on the server has been implemented in Java, as an "Action" of the Seam Framework[15], upon which KiWi is built.

The KQB interacts with this server-side Java code via AJAX (Asynchronous Java Script and XML), which makes the client-server communication between the two languages (Java and Java Script) possible.

The Seam action includes functions to provide the KQB with a list of all saved queries, to load one of those queries, to delete one, and to save a new query to the list.

Since the KQB supports full roundtripping, when saving a query, only the short, textual KWQL query is saved on the server, and upon loading, a KWQL query is parsed to visKWQL. This way, the amount of space needed to save even a large number of queries for a large number of users stays small, since no additional visual information needs to be saved.

[15] http://seamframework.org/

For query execution, the KQB will also have to communicate with the KiWi server, but since KWQL has not been implemented as of writing this, this functionality has not been included yet.

3.4.1 Java Script

While the support for HTML and CSS is similar in all browsers, the Java Script part of DHTML is somewhat problematic, since different browser vendors use different versions of the language, called either ECMA Script, JScript or Java Script.

Different vendors also provide different support to access a web site's structure (the Document Object Model) via Java Script, making it impossible to have the same code produce the same results in all browsers.

In the development of visKWQL and the KQB great care has therefore been taken to write functions which access the DOM, and others where necessary, with alternative code paths for different browsers, to enable the system to run on all common browsers.

The KQB has been tested with the current versions of the most common browsers (Internet Explorer 8, FireFox 3.0, Google Chrome 2.0, Safari 4, Opera 10.0).

While almost every browser generation also introduces a new iteration of the Java Script language, the use of the newest language features has been omitted in the KQB, to enable backwards-compatibility with older browsers. The KQB uses only features of Java Script 1.5 (or its equivalent ECMA Script 3 or JScript 5), which should allow it to run on earlier browser versions (Internet Explorer 6 and 7 and FireFox from version 1.0 [70]).

In recent browser generations, Java Script performance has seen considerable improvements [71], which have allowed to place user experience higher than performance considerations during the development of the KQB.

Thus the system runs smoothly, even when it executes much code, for example when the user is dragging something, which requires checks on every mouse movement, to provide instant user feedback like the changing of box border colors between green, red and orange when a box is dragged over another box where it can or can not be dropped.

However, to enable the system to run smoothly on older browsers with slower Java Script implementations, or on slower computers, the KQB code has been optimized for speed where possible, for example by introducing a tree like structure to hold box coordinates and types for the border color checks mentioned above, which allow them to be executed much faster than by querying the DOM on every mouse movement.

3.4.2 Roundtripping

The first direction of roundtripping, the translation from visKWQL to KWQL, can, thanks to the closeness of the visual to the textual representation, be seen as a serialization of the visual query. In most cases, the visual child-inclusion, which is the only real relationship between elements in visKWQL, results in a `qualifier:value` or `resource ( child1 child2 ...)` construct in KWQL, making the translation straightforward.

The other direction, the translation from KWQL to visKWQL, however, is more complex, and requires a KWQL parser.

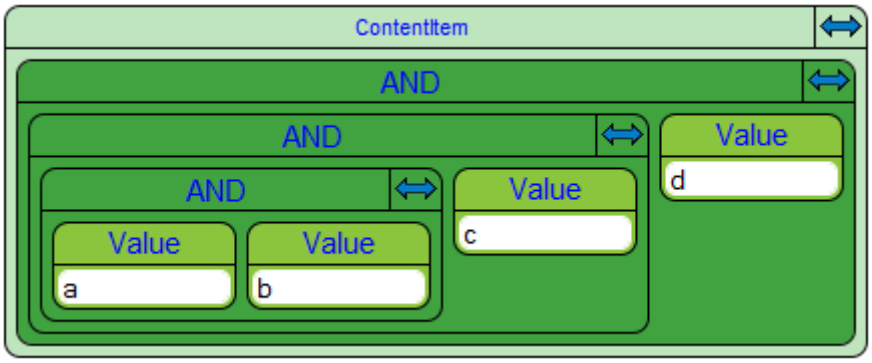
A first version of the parser has been written by hand. But since then the KWQL grammar has been formally specified in ANTLR[16] (ANother Tool for Language Recognition), a tool that allows the specification and checking of grammar rules, and acts as a parser generator that also supports Java Script, made it possible to use a generated parser.

A generated parser has the advantages over a handwritten one of providing better error checking, and, most importantly, allowed to quickly generate a new parser every time the KWQL grammar changed, which will also be helpful if KWQL should be changed or extended in the future.

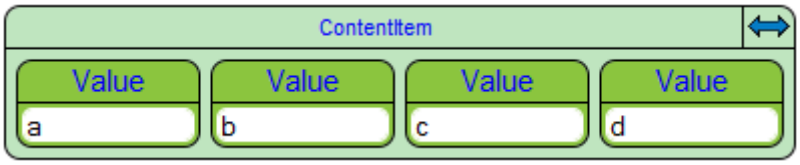
The generated parser creates an AST (Annotated Syntax Tree) from a textual KWQL query, a tree structure consisting of nodes representing different grammar rules or string values. This AST is then converted by the KWQL Query Builder into a visKWQL query.

It has been tried to make the code converting the AST to visKWQL as general as possible, but it still includes some parts very specific to KWQL.

A reason for non-general translation code is, that with specific checks on AST nodes, queries can be simplified during parsing to result in an easier to understand visual query.



(a) General parsing



(b) Parsing with query simplification

Figure 3.31: The query ci (a b c d)

Figure 3.31 demonstrates this simplification on the query `ci (a b c d)`.

The KWQL grammar parses connective operators into a binary AST sub-tree, with each node having two children, resulting in the visual query of Figure 3.31a, with three AND boxes connecting the four values. In many languages this would be the only way to translate, but in (vis)KWQL, the AND is implicit within a Resource, so that the four values can be direct children of the Content Item and the three AND boxes are unnecessary. The translation code of the KQB takes such language features into account, and removes the superfluous AND nodes during parsing, to create the simplified, but semantically equal, query of Figure 3.31b.

[16] http://www.antlr.org/

3.4.3 Customizability

An important aspect in the development of visKWQL and the KQB has been support for easy changeability and customizability.

KWQL itself was in development during the development of visKWQL, and might still change after the completion of the KQB prototype, so that care has been taken to write the KQB in a way that allows for later addition of new language constructs. To make this easier, all visKWQL elements (boxes) are defined in the same conceptual way, having data like name and type, and a list of all the types that are allowed as children or are compatible for a type switch.

Thus, when adding a new language element, like a new qualifier or operator, the only work that needs to be done is to define this element through the mentioned data, update the allowed type lists of other elements the new one is compatible with, and add a rule in each of the two translation routines for the new element, one for the direction from KWQL to visKWQL (if the new element is an operator, it is also necessary to generate a new parser from the updated grammar rules), and one for the translation from visKWQL to KWQL.

Secondly, all strings that appear in the KQB are collected in one Java Script source file[17], including the names of visKWQL elements like Qualifiers, the tooltips for all elements, error and warning messages, and hints.

This allows to quickly find and change any of those, and also allows the easy localization of the KQB, by simply translating those strings into a new language like German, without touching any of the KQB source code.

Finally the third, and most important, feature is visual customizability.

Apart from structure, all visual aspects of visKWQL and the KQB, including the colors and sizes of all elements, in addition to details like line widths, paddings or fonts, are defined in a CSS file, which allows for easy changing of any of those aspects.

The KQB prototype has been created with mostly green tones to resemble the green KiWi color theme, but for anyone familiar with CSS it is easy to quickly change this.

The KQB code also has no hard-coded values for element sizes, but retrieves them from the CSS, so that values like box size, line width or paddings can also easily be customized.

It should also be added here that the KQB prototype includes some CSS 3.0 instructions for a more visually appealing look, but these are not yet supported by all browsers.

Figure 3.32 shows the differences between the look of visKWQL queries on browsers without (Fig. 3.32a) and with (Fig. 3.32b) support for CSS 3.0 rounded edges.

[17] collecting them in an external text file is not possible, since, for internet security reasons, Java Script has no access to external files

(a) Internet Explorer, Opera

(b) FireFox, Safari

Figure 3.32: Browser display differences

Chapter 4

Conclusion

Over the past three decades, ever since with QBE the first visual language has started a whole field of research, the number of visual languages that have been designed or implemented has grown exceptionally large.

But even the relatively small number of examples that has been given in chapter 2 shows the wide variety of these languages, not only in their purpose, where they include data querying, programming and modeling, or their underlying textual languages, which include almost all common query and programming languages, but especially in their visual approaches, ranging from form-based to iconic to diagrammatically to the three dimensional.

While some visual languages, like UML, are used regularly today, others have not been much more than academic exercises.

One reason for that is too much abstraction, which made the languages easy to use, but sacrificed too much expressivity in the process, so that the resulting language had not much practical use.

Another reason is too little investment in user-friendliness, which resulted in visual languages that had high expressivity, but which used so many different visual syntax elements, including shapes, colors, line and arrow types, spatial arrangement rules, or other elements, that they were just different from their textual counterpart, but not easier to learn and use. Instead of keywords and textual syntax rules, a user had to spend a lot of time learning the meaning of visual elements to be able to use the language.

visKWQL was designed from the beginning to avoid both of these pitfalls. It has the full expressivity of the underlying textual language, but visually consists only of one type of element, boxes, and one type of relationship between those elements, child inclusion, giving it an easy to learn and understand visual syntax.

On the other hand, it could be argued that due to it using boxes to represent everything, and using text to indicate what exactly an individual box represents, it is a less visual language than one that uses no textual indicators at all.

However, for most users, who want to quickly create a meaningful query, it should be easier to create a visual query from boxes with textual labels that describe the box, than to first learn a larger number of different visual representations and their accompanying KWQL elements.

One problem of the nested boxes approach of visKWQL is that longer queries quickly

occupy a large amount of space.

While graph-based visual query languages often allow the user to spatially arrange different parts of the query, in visKWQL children are always arranged horizontally, making queries quickly grow in that direction.

Letting the user change the shape of the boxes, to also allow vertical arrangement of children would solve the problem of queries mostly growing horizontally.

Employing a zooming mechanism, like VXT does (see chapter 2.3.2), would also help visualizing the whole query at once, even if it is very large. Such a mechanism, at least in a visually appealing way, with smooth zooming, however, would be very difficult to realize in DHTML.

However, the chosen solution of collapsible boxes, allowing to hide parts of the query, is an acceptable compromise, which allows the user to work with large queries, but is also easy to use and to realize with current browser technology.

A last problem of visKWQL, and of most other visual languages, is that they provide little advantage to professional users, who have a very good understanding of the underlying language.

They know the syntax well enough to not require the additional help of visKWQL, in the form of error prevention and display, and will see no speed-up in visual query construction compared to typing a textual query.

However, for other users, the inexperienced and intermediate ones, visKWQL provides advantages that should motivate those users to visually construct queries, which would be possible to show in a future evaluation of the system in tests with such users.

Completely inexperienced users benefit the most, since the system shows them all the language elements they can use, and makes sure they use them in a correct way. It also provides them with tips and hints, so that they do not feel lost and abandon their attempt at creating an interesting query.

Intermediate users have an understanding of the different language elements and query construction in general. But they will still benefit from using the visual language, mostly from the error prevention and display features of the system, helping them to create the queries they want.

KWQL itself was designed with a smooth learning curve in mind, letting new users start with simple keywords, intermediate ones with queries involving KiWi's metadata, and leading to professional users constructing complex rules.

visKWQL also tries to provide a learning experience for the user. Its visual nature and the help it provides should motivate new users to try creating queries that are more involved than just simple keyword queries, and encourage advanced users to try creating rules that improve their work in KiWi.

Its roundtripping feature allows users who are already familiar with KWQL, but still want to use visKWQL, to quickly obtain visual versions of their textual queries, and to save time by editing different parts of a query in the different representations.

At the same time, roundtripping teaches those users who have no knowledge of KWQL the textual language too, since every change they make in the visual query is immediately displayed in its textual form.

visKWQL will thus hopefully be seen as a worthwhile tool by its intended users, and provide its part to KiWi's goal of user-centeredness by enriching KWQL in a user-friendly, visual way.

Bibliography

[1] S. Schaffert et al. The KiWi Vision: Collaborative Knowledge Management, powered by the Semantic Web, KiWi Public Deliverable D8.5. `http://wiki.kiwi-project.eu/multimedia/kiwi-pub:KiWi_D8.5_final.pdf`, 2009.

[2] F. Bry and K. Weiand. Flavors of KWQL, a Keyword Query Language for a Semantic Wiki. *Manuscript*, 2009.

[3] M. M. Zloof. Query-By Example: A Data Base Language. *IBM Systems Journal 16*, 1977.

[4] Unified Modelling Language Specification. `http://www.omg.org/spec/UML/2.2/`.

[5] M. Kindborg and K. McGee. Visual Programming with Analogical Representations: Inspirations from a Semiotic Analysis of Comics. *Journal of Visual Languages and Computing*, 2007.

[6] John McCarthy. The Sayings of John McCarthy. `http://www-formal.stanford.edu/jmc/sayings.html`.

[7] M. M. Burnett. Visual Programming. *Encyclopedia of Electrical and Electronics Engineering (John G. Webster, ed.), John Wiley I& Sons Inc.*, 1999.

[8] B. A. Price, I. S. Small, and R. M. Baecker. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing 4(3)*, 1993.

[9] T. Catarci, MF. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *Journal of visual languages and computing*, 1997.

[10] T. Catarci and G. Santucci. Are Visual Query Languages Easier to Use than Traditional Ones? An Experimental Proof. *Proceedings of the HCI'95 conference on People and computers X*, 1995.

[11] M. Angelaccio, T. Catarci, and G. Santucci. QBD*: A graphical query language with recursion. *IEEE Transactions on Software Engineering*, 1990.

[12] M. Angelaccio, T. Catarci, and G. Santucci. QBD*: A Fully Visual Query System. *Journal on Visual Languages and Computing*, 1990.

[13] Wikipedia: Sankey diagram. `http://en.wikipedia.org/wiki/Sankey_diagram`, retrieved on August 10, 2009.

[14] International Organization for Standardization. Information processing – Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts, 1985.

[15] M. Laguna and J. Marklund. Business Process Modeling, Simulation, and Design, 2004.

[16] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN Notices, Volume 8, Issue 8*, 1973.

[17] P. P. S. Chen. The Entity-Relationship Model–Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1976.

[18] R. Elsmari and S. B. Navathe. Fundamentals of Database Systems, Fourth Edition, Chapter 3. 2004.

[19] D. D. Hils. Visual Languages and Computing Survey: Data Flow Visual Programming Languages. *Journal of Visual Languages and Computing 3/1992*, 1992.

[20] R. Smith. The alternate reality kit: An animated environment for creating interactive simulations. *Proceedings of the 1986 IEEE Workshop on Visual Languages*, 1986.

[21] D. C. Smith, A. Cypher, and J. Spohrer. KidSim: Programming Agents Without a Programming Language. *Communications of the ACM*, 1994.

[22] M. Boshernitsan and M. Downes. Visual programming languages: A Survey, 2004.

[23] T. F. Materson and R. M. Meyer. SIVIL: a true visual programming language for students. *Journal of Computing Sciences in Colleges, Vol. 16 Issue 1*, 2001.

[24] E. Pietriga, J. Y. Vion-Dury, and V. Quint. VXT: a visual approach to XML transformations. *Proceedings of the ACM Symposium on Document engineering*, 2001.

[25] J. Clark et al. XSL Transformations (XSLT), Version 1.0, W3C Recommendation. `http://www.w3.org/TR/xslt`, 1999.

[26] Yahoo! Pipes Documentation. `http://pipes.yahoo.com/pipes/docs`, 2009.

[27] The Official Yahoo! Pipes Blog: Introducing Pipes. `http://blog.pipes.yahoo.net/2007/02/07/introducing-pipes/`, 2007.

[28] S. Kittoli et al. JBoss Seam 2.2.0.GA Documentation.

[29] M. Najork and S. Kaplan. The Cube Language. *IEEE Workshop on Visual Languages*, 1991.

[30] T. Oshiba and J. Tanaka. "3D-PP": Visual Programming System with Three-Dimensional Representation. *Proceedings of International Symposium on Future Software Technology*, 1999.

[31] A. Scharnofske, U. W. Lipeck, and M. Gertz. SubQueryByExample: Eine orthogonale Erweiterung von QBE. *Springer: Datenbanksysteme in Büro, Technik und Wissenschaft*, 1997.

[32] N. A. Lorentzos and K. A. Dondis. Query by Example for Nested Tables. *Lecture notes in computer science - Springer*, 1998.

[33] I. M. R. E. Filha, A. H. F. Laender, and A. S. da Silva. Querying semistructured data by example: the QSByE interface. *Workshop on Information Integration on the Web*, 2001.

[34] S. Cohen, Y. Kanza, Y. Kogan, W. Nutt, et al. Equix - Easy Querying in XML Databases. *2nd ACM SIGMOD Int. Workshop on The Web and Databases*, 1999.

[35] S. Cohen, Y. Kanza, Y. Kogan, W. Nutt, et al. EquiX - a search and query language for XML. *Journal of the American Society for Information Science and Technology*, 2002.

[36] K. D. Munroe and Y. Papakonstantinou. BBQ: A visual interface for integrated browsing and querying of XML. *Proceedings of the Fifth Working Conference on Visual Database Systems: Advances in Visual Information Management*, 2000.

[37] M. Erwig. XML queries and transformations for end users, 2000.

[38] M. Erwig. A visual language for XML. *IEEE Symposium on Visual Languages*, 2000.

[39] I.F. Cruz, A.O. Mendelzon, and P.T. Wood. A graphical query language supporting recursion. *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, 1987.

[40] M.P. Consens and A.O. Mendelzon. GraphLog: a visual formalism for real life recursion. *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1990.

[41] J. Paredaens, J. Van den Bussche, M. Andries, et al. An overview of GOOD. *ACM SIGMOD Record*, 1992.

[42] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object model for database end-user interfaces. *ACM SIGMOD Record*, 1990.

[43] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. A graph-oriented object database model. *IEEE Transactions on Knowledge and Data Engineering*, 1994.

[44] J. Paredaens, P. Peelman, and L. Tanca. G-log: A graph-based query language. *IEEE Transactions on Knowledge and Data Engineering*, 1995.

[45] S. Comai, E. Damiani, and L. Tanca. The WG-Log System: Data Model and Semantics.

[46] S. Comai. Graphical Query Languages for Semi-structured Infomation. *International Conference on Extending Database Technology*, 1999.

[47] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: a graphical language for querying and restructuring XML documents. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 1999.

[48] S. Comai, E. Damiani, and P. Fraternali. Computing graphical queries over XML data. *ACM Transactions on Information Systems (TOIS)*, 2001.

[49] D. Chamberlin et al. XQuery 1.0: An XML Query Language, W3C Recommendation. `http://www.w3.org/TR/xquery/`, 2007.

[50] D. Braga and A. Campi. A graphical environment to query XML data with XQuery. *Proceedings of the Fourth International Conference on Web Information Systems Engineering*, 2003.

[51] S. Berger. Conception of a Graphical Interface for Querying XML, 2003.

[52] S. Berger, F. Bry, S. Schaffert, and C. Wieser. Xcerpt and visXcerpt: From Pattern-Based to Visual Querying of XML and Semistructured Data. *Proceedings of the 29th international conference on Very large data bases - Volume 29*, 2003.

[53] S. Berger, F. Bry, and S. Schaffert. A Visual Language for Web Querying and Reasoning. *Proceedings of Workshop on Principles and Practice of Semantic Web Reasoning (ICLP03)*, 2003.

[54] S. Berger, F. Bry, T. Furche, and C. Wieser. Visual Languages: A Matter of Style, 2007.

[55] F. Bry and S. Schaffert. A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML, 2002.

[56] A. Russell, P. R. Smart, D. Braines, and N. R. Shadbolt. NITELIGHT: A Graphical Tool for Semantic Query Construction. *Semantic Web User Interaction Workshop (SWUI 2008)*, 2008.

[57] B. McBride et al. RDF Primer - W3C Recommendation. `http://www.w3.org/TR/REC-rdf-syntax/`, 2004.

[58] F. Frasincar U. Kaymak F. Hogenboom, V. Milea. RDF-GL: A SPARQL-Based Graphical Query Language for RDF. `http://people.few.eur.nl/frasincar/papers/EWI2009/ewi2009.pdf`, 2009.

[59] M. Kuntz and R. Melchert. Pasta-3's graphical query language: direct manipulation cooperative queries, full expressive power. *Proceedings of the 15th international conference on Very large data bases table of contents*, 1989.

[60] S. Dar, N. H. Gehani, H. V. Jagadish, and J. Srinivasan. Queries in an Object-Oriented Graphical Interface. *Journal of Visual Languages and Computing*, 1995.

[61] M. J. Carey, L. M. Haas, V. Maganty, and J. H. Williams. PESTO: An Integrated Query/Browser for Object Databases. *Proceedings of the 22th International Conference on Very Large Data Bases*, 1996.

[62] R. King and S. Melville. SKI: A Semantic Knowledgeable Interface. *Proceedings of the 10th International Conference on Very Large Databases*, 1984.

[63] M. Moscarini, F. Fontana, and M. Valeri. IVQS: a visual client system to query remote multimedia databases through the web with an iconic interface. *1. united GI conference*.

[64] S. Schaffert. IkeWiki: A Semantic Wiki for Collaborative Knowledge Management. *1st International Workshop on Semantic Technologies in Collaborative Applications STICA 06, Manchester, UK*, 2006.

[65] P. Reiser. SunSpace Blog. `http://blogs.sun.com/peterreiser/category/SunSpace`, 2009.

[66] J. Bailey, F. Bry, T. Furche, and S. Schaffert. Web and Semantic Web Query Languages: A Survey. *Reasoning Web Proceedings*, 2005.

[67] T. Furche, B. Linse, F. Bry, D. Plexousakis, and G. Gottlob. RDF Querying: Language Constructs and Evaluation Methods Compared. *Reasoning Web Proceedings*, 2006.

[68] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF - W3C Recommendation. `http://www.w3.org/TR/rdf-sparql-query/`, 2008.

[69] K. Weiand and F. Bry. Reasoning and Querying - Concept and Model, KiWi Public Deliverable D2.2. `http://www.kiwi-project.eu/index.php/publications/33-deliverables`, 2009.

[70] J. Resig. Versions of JavaScript. `http://ejohn.org/blog/versions-of-javascript/`, 2008.

[71] celtic kane blog. Web Browser Javascript Benchmark. `http://celtickane.com/labs/web-browser-javascript-benchmark`, 2009.